

TURING

图灵计算机科学丛书

PEARSON  
Addison  
Wesley

# 数据结构与算法分析

## C++描述

(第3版)

**Data Structures and Algorithm Analysis in C++**  
**Third Edition**

[美] Mark Allen Weiss 著  
张怀勇 等译 刘田 审校



人民邮电出版社  
POSTS & TELECOM PRESS

# 版 权 声 明

Authorized translation from the English language edition, entitled *Data Structures and Algorithm Analysis in C++*, Third Edition, 0321375319 by Mark Allen Weiss, published by Pearson Education, Inc., publishing as Addison-Wesley, Copyright © 2006 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS Copyright © 2006.

本书中文简体字版由Pearson Education Asia Ltd.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。版权所有，侵权必究。

## 译者序

数据结构与算法分析是计算机专业最重要的基础课之一，在计算机科学及相关领域有广泛应用。本书的原文是国外数据结构与算法分析课程的经典教材，有诸多版本在国内引进出版，包括C语言版、Java语言版等，对国内的数据结构与算法分析领域的教学工作有很大的积极影响。

书中论述了数据结构（大量数据的组织方法）以及算法分析（算法运行时间的估算）。书中采用当前流行的面向对象语言C++来描述数据结构和算法，并提供了大部分算法的C++程序和伪代码例程。本书不但详细介绍了当前流行的论题和新的变化，讨论了算法设计技巧，而且在研究算法的性能、效率以及对运行时间分析的基础上考查了一些高级数据结构，从历史的角度和近年的发展对数据结构的活跃领域进行了简要的概括。

本书可作为高级数据结构课程或研究生一年级算法分析课程的教材。本书在教会学生良好的程序设计技巧的同时让学生具备算法分析能力，使得他们能够开发高效的程序。

译者在翻译过程中，参考了天津师范大学冯舜玺老师翻译的本书Java版的译文，而且得到冯老师的诸多帮助，使翻译工作得以尽快完成，在此深表谢意。特别感谢北京大学刘田老师，译稿成文过程中得到他的悉心审阅，提高了本书的翻译质量。我还要借此机会感谢本书的合译者吴怡，感谢她出色的翻译工作。我还想感谢人民邮电出版社图灵公司的编辑们，感谢他们为使最后的散稿成书所做的出色工作。

张怀勇  
2006年7月



# 前言

---

## 目的/目标

本书全面论述了数据结构和算法分析，即组织大量数据的方法和对算法运行时间的估计。随着计算机的速度越来越快，对于能够处理大量输入数据的程序的需求变得日益迫切。具有讽刺意味的是，由于在输入量很大时程序的效率明显降低，因此这又要求更加关注效率问题。通过在实际编程之前对算法进行分析，学生可以确定一个特定的解法是否可行。例如，在本书中学生可看到一些特定的问题，并了解精心的实现如何能够把处理大量数据的时间从16年减至不到1秒。因此，本书中论述的算法和数据结构均进行了运行时间方面的分析。在某些情况下，还研究了影响实现运行时间的一些微小细节。

一旦确定了解法，接着就要编写程序。随着计算机功能的日益强大，它们必须解决的问题也越来越大，越来越复杂，这就要求开发更加复杂的程序。本书的目的是在教会学生使用良好的程序设计技巧的同时让学生具备算法分析能力，使得他们开发的这类程序具有最高效率。

本书适用于本科生的高级数据结构课程或是研究生的算法分析课程。使用本书的学生应该具有中等程度的程序设计方面的知识，包括指针、递归和面向对象程序设计，还要具有离散数学的某些知识。

---

## 方法

虽然本书中的内容大部分都是与语言无关的，但是，程序设计还是需要使用某种特定的语言。正如书名指出的，我们为本书选择了C++。

C++已经成为系统编程的主流语言。除了修正了许多C语言的语法方面的缺陷之外，C++还提供了直接结构（类和模板）来实现抽象数据类型的通用数据结构。

撰写本书最困难的部分是确定C++在书中所占的比例。使用太多的C++的特性将使教材变得很晦涩，使用得太少又会和使用支持类的C语言撰写的教材区别不大。

我们采用的方法是以基于对象的方法来阐述文中的内容。这样，本书中几乎没有用到继承性。我们采用类模板来描述通用数据结构。通常情况下尽量避免使用深奥的C++特性，而是使用标准C++中的vector和string类。通过采用C++的现代特性来取代在本书第1版中所用的次级特性简化了很多代码。本书前几版已经实现了类模板，方法是将类模板的接口与其实现分离开。毫无疑问，这是一种好方法，但是这种方法暴露出了编译的问题，读者事实上很难利用这些代码。本版中，在线的源代码将类模板作为一个单元来表示，而不再将接口和实现分离开。本书第1章对书中所用到的C++特性进行了介绍，并阐述了对类模板的处理方法。附录描述了如何重写类模板，用于分离编译。



以Java和C++两种语言描述的数据结构的完全版在因特网上可以得到。我们使用类似的编码习惯使得这两种语言间的相似性表现得更加明显。

---

## 第3版中的主要变化

---

第3版中包含了大量的错误修正，并且书中的大部分章节都经过了修订以提高其可读性。另外，本版还有以下几方面的变化：

- 书中的所有代码都做了更新以适应现代的C++特性。
- 第3章进行了大量的修订，并且讨论了标准vector和list类的使用及其实现。标准vector类在其他数据结构的实现中被广泛使用。
- 第4章修订后包含了关于set和map类的讨论，并通过一个扩展的例子介绍了它们在有效算法设计中的应用。在第9章中也包含了一个使用标准map类来实现最短路径算法的例子。
- 第7章讨论了标准sort算法，其中包括一个关于实现重载的标准sort算法所涉及的技巧。
- 书中提供的源代码都已经进行了简化，从而避免了与类模板的分离编译有关的复杂语法。修订后的代码可以使读者更专心于算法本身，而不是过多地关注C++。

---

## 内容概述

---

第1章包含离散数学和递归的一些内容。我相信熟悉递归的唯一办法是反复不断地看一些好的用法。因此，除第5章外，递归遍布本书每一章的例子中。第1章还介绍了一些C++的内容，包括C++基础知识的回顾以及C++类设计中模板和重要结构的讨论。

第2章讨论算法分析。这一章阐述了渐近分析和它的主要弱点。这里提供了许多例子，包括对对数运行时间的深入解释。通过直观地把一些简单递归程序转变成迭代程序，对它们进行分析。这一章还介绍了更复杂的分治程序，不过有些分析（求解递归关系）要到第7章再详细进行。

第3章包括表、栈和队列。这一章较之以前的版本进行了大量的修订。现在包含了关于STL vector和list类的讨论，包括有关迭代的内容，并且提供了STL vector和list类的重要子集的实现。

第4章讨论树，重点在查找树，包括外部查找树（B树）。UNIX文件系统和表达式树是作为例子介绍的。本章还介绍了AVL树和伸展树。查找树实现细节的更详细讨论可在第12章找到。树的另外一些内容，如文件压缩和博弈树，延迟到第10章讨论。外部介质上的数据结构作为几章中的最后论题来考虑。本版中新增的部分是对STL set和map类的讨论，包括一个讲解了如何使用三个分离的图来高效率地解决问题的例子。

第5章相对较短，主要讨论散列表。这里进行了某些分析，本章末尾讨论了可扩展散列。

第6章是关于优先队列的。这一章还讲解了二叉堆，还有一些附加内容，论述优先队列某些理论上很有趣的实现方法。斐波那契堆在第11章讨论，配对堆在第12章讨论。

第7章是排序。它是关于编程细节和分析的非常特殊的一章，讨论并比较了所有重要的通用排序算法。对插入排序、希尔排序、堆排序以及快速排序这四种算法进行了详细的分析。这一章末尾还讨论了外部排序。

第8章讨论不相交集算法并证明其运行时间。这一章短且特殊，如果不讨论Kruskal算法则这一章可跳过。

第9章讲授图论算法。图论算法之所以重要，不仅因为它们在实践中频繁出现，而且因为它

们的运行时间特别依赖于数据结构的恰当使用。实际上，所有标准算法都是和相应的数据结构、伪代码以及运行时间的分析一起介绍的。为把这些问题放在一个适当的环境下讨论，书中提供了对复杂性理论（包括NP完全性和不可判定性）的简短讨论。

第10章通过考查一般的问题求解技巧来介绍算法设计。这一章含有大量的实例。这里及后面各章使用了伪代码，使学生对一个示例算法的理解不受具体实现细节的困扰。

第11章处理摊还分析。对第4章和第6章的三种数据结构以及本章介绍的斐波那契堆进行了分析。

第12章讨论查找树算法、 $k$ -d树和配对堆。不同于其他各章，这一章给出了查找树和配对堆完全的仔细的实现。材料的安排使得教师可以把一些内容纳入到其他各章的讨论之中。例如，第12章中的自顶向下红黑树可以和（第4章的）AVL树一起讨论。

第1章~第9章为大多数一学期的数据结构课程提供了足够的材料。如果时间允许，第10章也可以包括进来。研究生的算法分析课程可以使用第7章到第11章的内容。在第11章分析的高级数据结构可以很容易地在前面各章中查到。第9章中对NP完全性的讨论太过简短，以至于不能用于算法分析课程。可以使用论述NP完全性的其他书籍来补充本书的这部分不足。

---

## 练习

每章末尾提供的练习与正文中讲授内容的顺序相匹配。最后的练习是把一章作为一个整体来处理而不是针对特定的一节来考虑的。较困难的练习标有一个星号，更难的练习标有两个星号。

---

## 参考文献

参考文献位于每章的最后。一般说来，这些参考文献或者是历史性的，代表着书中材料的原始来源，或者阐述对书中给出的结果的扩展和改进。有些文献提供了一些习题的解法。

---

## 补充材料

所有的读者都可以在图灵网站[www.turingbook.com](http://www.turingbook.com)或[www.aw.com/cssupport](http://www.aw.com/cssupport)网站得到下面的补充材料：

- 书中例子的程序代码。

此外，以下材料可从Addison-Wesley的教师资源中心（[www.aw.com/irc](http://www.aw.com/irc)）获得，但是这部分内容仅针对有资质的教师。教师可以访问Addison-Wesley的教师资源中心或向当地的Addison-Wesley代表申请下面这些资料。

- 书中部分练习的答案。
- 书中的图。

---

## 致谢

在本书各个版本的准备过程中，我得到许多人的帮助。有些人在本书的其他版本中提到过，谢谢他们每一位。



同往常一样，Addison-Wesley的专家们使得本书的写作过程更加轻松。我愿意借此机会感谢本书的编辑Michael Hirsch和文字编辑Marilyn Lloyd。我还想感谢Paul Anagnostopoulos和他在Windfall Software的同事，感谢他们的出色工作使最后的书稿成书。特别感谢我的爱妻Jill，感谢她所做的一切。

最后，我还想感谢广大的读者，他们发来电子邮件并指出前面各版中的一些错误和矛盾之处。我的网页<http://www.cis.fiu.edu/~weiss>将包含更新的（C++的、C的以及Java的）源代码、勘误表以及指向提交问题报告的一个链接。

Mark Allen Weiss  
于佛罗里达州迈阿密

# 目 录

第1章 引论	1	1.7.2 operator[]	28
1.1 本书讨论的内容	1	1.7.3 析构函数、复制赋值和复制构造函数	29
1.2 数学知识复习	2	小结	29
1.2.1 指数	2	练习	29
1.2.2 对数	2	参考文献	30
1.2.3 级数	3	第2章 算法分析	32
1.2.4 模运算	4	2.1 数学基础	32
1.2.5 证明方法	4	2.2 模型	34
1.3 递归的简单介绍	6	2.3 要分析的问题	34
1.4 C++类	9	2.4 运行时间计算	36
1.4.1 基本class语法	9	2.4.1 一个简单的例子	37
1.4.2 特别的构造函数语法与访问函数	10	2.4.2 一般法则	37
1.4.3 接口与实现的分离	11	2.4.3 最大子序列和问题的解	39
1.4.4 vector和string	13	2.4.4 运行时间中的对数	44
1.5 C++细节	14	2.4.5 检验你的分析	46
1.5.1 指针	14	2.4.6 分析结果的准确性	47
1.5.2 参数传递	15	小结	48
1.5.3 返回值传递	16	练习	48
1.5.4 引用变量	16	参考文献	52
1.5.5 三大函数：析构函数、复制构造函数和operator=	17	第3章 表、栈和队列	53
1.5.6 C风格的数组和字符串	20	3.1 抽象数据类型(ADT)	53
1.6 模板	21	3.2 表ADT	53
1.6.1 函数模板	22	3.2.1 表的简单数组实现	54
1.6.2 类模板	22	3.2.2 简单链表	54
1.6.3 Object、Comparable和例子	24	3.3 STL中的向量和表	55
1.6.4 函数对象	25	3.3.1 迭代器	56
1.6.5 类模板的分离编译	26	3.3.2 示例：对表使用erase	57
1.7 使用矩阵	27	3.3.3 const_iterator	58
1.7.1 数据成员、构造函数和基本访问函数	27	3.4 向量的实现	59
		3.5 表的实现	63



3.6 栈ADT	71	参考文献	135
3.6.1 栈模型	71	第5章 散列	137
3.6.2 栈的实现	72	5.1 基本思想	137
3.6.3 应用	72	5.2 散列函数	137
3.7 队列ADT	78	5.3 分离链接法	139
3.7.1 队列模型	78	5.4 不使用链表的散列表	142
3.7.2 队列的数组实现	78	5.4.1 线性探测	142
3.7.3 队列的应用	80	5.4.2 平方探测	144
小结	81	5.4.3 双散列	148
练习	81	5.5 再散列	148
第4章 树	85	5.6 标准库中的散列表	150
4.1 预备知识	85	5.7 可扩散列	151
4.1.1 树的实现	86	小结	153
4.1.2 树的遍历及应用	87	练习	154
4.2 二叉树	90	参考文献	156
4.2.1 实现	90	第6章 优先队列(堆)	158
4.2.2 一个例子——表达式树	91	6.1 模型	158
4.3 查找树ADT——二叉查找树	93	6.2 一些简单的实现	159
4.3.1 contains	94	6.3 二叉堆	159
4.3.2 findMin和findMax	96	6.3.1 结构性质	159
4.3.3 insert	97	6.3.2 堆序性质	160
4.3.4 remove	98	6.3.3 基本的堆操作	161
4.3.5 析构函数和复制赋值操作符	99	6.3.4 堆的其他操作	164
4.3.6 平均情况分析	99	6.4 优先队列的应用	167
4.4 AVL树	102	6.4.1 选择问题	167
4.4.1 单旋转	104	6.4.2 事件模拟	168
4.4.2 双旋转	106	6.5 $d$ 堆	169
4.5 伸展树	111	6.6 左式堆	170
4.5.1 一个简单的想法(不能直接使用)	112	6.6.1 左式堆性质	170
4.5.2 伸展	113	6.6.2 左式堆操作	171
4.6 树的遍历	118	6.7 斜堆	175
4.7 B树	119	6.8 二项队列	177
4.8 标准库中的set和map	123	6.8.1 二项队列结构	177
4.8.1 set	123	6.8.2 二项队列操作	178
4.8.2 map	124	6.8.3 二项队列的实现	180
4.8.3 set和map的实现	125	6.9 标准库中的优先队列	183
4.8.4 使用几个map的例子	126	小结	185
小结	130	练习	185
练习	130	参考文献	189

第7章 排序 .....	191	8.2 动态等价性问题 .....	231
7.1 预备知识 .....	191	8.3 基本数据结构 .....	233
7.2 插入排序 .....	192	8.4 灵巧求并算法 .....	235
7.2.1 算法 .....	192	8.5 路径压缩 .....	237
7.2.2 插入排序的STL实现 .....	192	8.6 按秩求并和路径压缩的最坏情形 .....	239
7.2.3 插入排序的分析 .....	194	8.7 一个应用 .....	243
7.3 一些简单排序算法的下界 .....	194	小结 .....	245
7.4 谢尔排序 .....	195	练习 .....	245
7.5 堆排序 .....	198	参考文献 .....	246
7.6 归并排序 .....	200	第9章 图论算法 .....	248
7.7 快速排序 .....	204	9.1 若干定义 .....	248
7.7.1 选取枢纽元 .....	206	9.2 拓扑排序 .....	250
7.7.2 分割策略 .....	206	9.3 最短路径算法 .....	252
7.7.3 小数组 .....	208	9.3.1 无权最短路径 .....	254
7.7.4 实际的快速排序例程 .....	208	9.3.2 Dijkstra算法 .....	257
7.7.5 快速排序的分析 .....	209	9.3.3 具有负边值的图 .....	262
7.7.6 选择问题的线性期望时间算法 .....	213	9.3.4 无环图 .....	263
7.8 间接排序 .....	213	9.3.5 所有顶点对的最短路径 .....	265
7.8.1 <code>vector&lt;Comparable*&gt;</code> 不运行 .....	215	9.3.6 最短路径举例 .....	266
7.8.2 智能指针类 .....	216	9.4 网络流问题 .....	267
7.8.3 重载 <code>operator&lt;</code> .....	217	9.5 最小生成树 .....	271
7.8.4 使用“*”解引用指针 .....	217	9.5.1 Prim算法 .....	272
7.8.5 重载类型转换操作符 .....	217	9.5.2 Kruskal算法 .....	273
7.8.6 随处可见的隐式类型转换 .....	217	9.6 深度优先搜索的应用 .....	275
7.8.7 双向隐式类型转换会导致歧义 .....	218	9.6.1 无向图 .....	276
7.8.8 指针减法是合法的 .....	218	9.6.2 双连通性 .....	276
7.9 排序算法的一般下界 .....	218	9.6.3 欧拉回路 .....	279
7.10 桶排序 .....	220	9.6.4 有向图 .....	282
7.11 外部排序 .....	220	9.6.5 查找强分支 .....	283
7.11.1 为什么需要新算法 .....	220	9.7 NP完全性介绍 .....	284
7.11.2 外部排序模型 .....	221	9.7.1 难与易 .....	285
7.11.3 简单算法 .....	221	9.7.2 NP类 .....	286
7.11.4 多路合并 .....	222	9.7.3 NP完全问题 .....	286
7.11.5 多相合并 .....	223	小结 .....	288
7.11.6 替换选择 .....	223	练习 .....	288
小结 .....	224	参考文献 .....	293
练习 .....	225	第10章 算法设计技巧 .....	296
参考文献 .....	229	10.1 贪心算法 .....	296
第8章 不相交集类 .....	231	10.1.1 一个简单的调度问题 .....	297
8.1 等价关系 .....	231	10.1.2 赫夫曼编码 .....	299



10.1.3 近似装箱问题 .....	303	11.4 斐波那契堆 .....	361
10.2 分治算法 .....	310	11.4.1 切除左式堆中的结点 .....	362
10.2.1 分治算法的运行时间 .....	310	11.4.2 二项队列的懒惰合并 .....	364
10.2.2 最近点问题 .....	312	11.4.3 斐波那契堆操作 .....	366
10.2.3 选择问题 .....	315	11.4.4 时间界的证明 .....	367
10.2.4 一些算术问题的理论改进 .....	317	11.5 伸展树 .....	368
10.3 动态规划 .....	320	小结 .....	371
10.3.1 用表代替递归 .....	320	练习 .....	371
10.3.2 矩阵乘法的顺序安排 .....	322	参考文献 .....	372
10.3.3 最优二叉查找树 .....	325	第12章 高级数据结构及其实现 .....	374
10.3.4 所有点对最短路径 .....	327	12.1 自顶向下伸展树 .....	374
10.4 随机化算法 .....	329	12.2 红黑树 .....	379
10.4.1 随机数生成器 .....	330	12.2.1 自底向上插入 .....	380
10.4.2 跳跃表 .....	333	12.2.2 自顶向下红黑树 .....	381
10.4.3 素性测试 .....	335	12.2.3 自顶向下删除 .....	385
10.5 回溯算法 .....	337	12.3 确定性跳跃表 .....	387
10.5.1 公路收费点重建问题 .....	337	12.4 AA树 .....	392
10.5.2 博弈 .....	341	12.5 treap树 .....	396
小结 .....	346	12.6 $k$ -d树 .....	399
练习 .....	346	12.7 配对堆 .....	401
参考文献 .....	351	小结 .....	405
第11章 摊还分析 .....	355	练习 .....	406
11.1 一个无关的智力问题 .....	355	参考文献 .....	408
11.2 二项队列 .....	356	附录 类模板的分离编译 .....	411
11.3 斜堆 .....	359	索引 .....	414

# 引 论

**本**章阐述本书的目的和目标，并且简要复习离散数学和程序设计的一些概念。我们将要了解程序在合理范围内的大规模输入情况下运行性能与其在中等规模输入下的运行性能同等重要。

- 概括本书其余部分所需要的基本的数学基础。
- 简要复习一下递归。
- 概括用于本书的C++语言的一些重要特点。

## 1.1 本书讨论的内容

设有一组 $N$ 个数，要确定其中第 $k$ 个最大者。这称为**选择问题**（selection problem）。对于大多数学习过一两门程序设计课程的学生来说，编写一个解决这种问题的程序不会有什么困难。“显而易见的”解决方法是相当多的。

该问题的一种解法就是将这 $N$ 个数读进一个数组中，再通过某种简单的算法，比如冒泡排序法，以递减顺序将数组排序，然后返回位置 $k$ 上的元素。

更好一点的算法可以先把前 $k$ 个元素读入数组并（以递减的顺序）对其排序。接着，将剩下的元素逐个读入。当读到一个新元素时，如果该元素小于数组中的第 $k$ 个元素则忽略之，否则就将其放到数组中的正确位置上，同时将数组中的一个元素挤出数组。当算法终止时，返回第 $k$ 个位置上的元素作为答案。

这两种算法的编码都很简单，建议读者试一试。此时读者自然要问：哪个算法更好？更重要的是，是否两个算法都足够好？使用含有1000万个元素的随机文件，在 $k=5\,000\,000$ 的条件下进行模拟发现，两个算法都不能在合理的时间内结束；每种算法都需要计算机处理若干天才能终止（尽管最终给出了正确答案）。在第7章讨论的另一种算法在1s左右就给出答案。因此，虽然所提出的两个算法都是正确的，但是都不能看作是好算法，因为相对于第三种算法能够在合理的时间内处理的输入规模而言，这两种算法都是完全不实用的。

第二个问题是求解一个流行的字谜游戏。输入由一个二维字母数组和一个单词表组成。目标是要找出字谜中的单词。这些单词可能是水平、垂直或沿任何对角线方向放置的。例如，图1-1所示的字谜由单词this、two、fat和that组成。单词this从第一行第一列的位置（即(1, 1)）处开始，并延伸至(1, 4)，单词two从(1, 1)到(3, 1)，fat从(4, 1)到(2, 3)，而that则从(4, 4)到(1, 1)。

同样，现在至少有两种显而易见的算法可以求解这个问题。对列出的每个单词，检查每一个有序三元组（行，列，方向）来看看这个单词是否存在。这需要大量嵌套的for循环，但基本上是显而易见的算法。



	1	2	3	4
1	t	h	i	s
2	w	a	t	s
3	o	a	h	g
4	f	g	d	t

图1-1 字谜示例

或者这样，对于每一个在字谜界限以内的有序四元组（行，列，方向，字符数），检查所指的单词是否在单词表中。同样，这导致大量嵌套的for循环。如果任意单词中的最大字符数已知，就有可能节省一些时间。

上述两种解法都相当容易编码，并且都能求解通常在杂志上发表的许多实际的字谜游戏。这些字谜通常有16行、16列以及40个左右的单词。然而，假设我们把字谜变成只给出谜板而单词表基本上是一本英语词典，则使用上面提出的两种解法来解决这个问题都需要相当可观的时间，因此这两种方法都是不可接受的。不过，即使单词表很大，这样的问题还是有可能在几秒钟内解决的。

在许多问题中，写出一个可以工作的程序是不够的。如果这个程序是在大规模的数据集上运行，那么运行时间就成了问题。在本书中我们将看到，对于大规模的输入，如何估计程序的运行时间，更重要的是，弄清如何在尚未具体编码的情况下比较两个程序的运行时间。我们还将看到提高程序的运行速度以及确定程序瓶颈的技巧。这些技巧将使我们能够找到需要着重优化的那些代码段。

## 1.2 数学知识复习

2 本节列出一些需要记住或是能够推导出的基本公式，并且复习基本的证明方法。

### 1.2.1 指数

$$\begin{aligned}
 X^A X^B &= X^{A+B} \\
 \frac{X^A}{X^B} &= X^{A-B} \\
 (X^A)^B &= X^{AB} \\
 X^N + X^N &= 2X^N \neq X^{2N} \\
 2^N + 2^N &= 2^{N+1}
 \end{aligned}$$

### 1.2.2 对数

在计算机科学中，所有的对数都是以2为底的，除非另有声明。

**定义1.1**  $X^A = B$ 当且仅当  $\log_X B = A$ 。

由该定义可以得到下面几个等式。

**定理1.1**  $\log_A B = \frac{\log_C B}{\log_C A}$ ； $A, B, C > 0$ ， $A \neq 1$ 。

**证明** 令  $X = \log_C B$ ， $Y = \log_C A$ ，以及  $Z = \log_A B$ 。于是，由对数的定义得  $C^X = B$ ， $C^Y = A$ 以及  $A^Z = B$ 。联合这三个等式则有  $B = C^X = (C^Y)^Z$ 。因此， $X = YZ$ ，这意味着  $Z = X/Y$ ，定理得证。 ■

**定理1.2**  $\log AB = \log A + \log B$ ;  $A, B > 0$ 。

**证明** 令  $X = \log A$ ,  $Y = \log B$ ,  $Z = \log AB$ 。此时由于假设默认的底为2, 所以  $2^X = A$ ,  $2^Y = B$ ,  $2^Z = AB$ 。联合最后的三个等式则有  $2^X 2^Y = AB = 2^Z$ 。因此  $X + Y = Z$ , 这就证明了该定理。 ■

其他一些有用的公式如下, 它们都能够用类似的方法推导出来。

$$\log A/B = \log A - \log B$$

$$\log(A^B) = B \log A$$

$$\log X < X \text{ 对所有的 } X > 0 \text{ 成立}$$

$$\log 1 = 0, \log 2 = 1, \log 1024 = 10, \log 1\,048\,576 = 20$$

3

### 1.2.3 级数

最容易记忆的公式是

$$\sum_{i=0}^N 2^i = 2^{N+1} - 1$$

和

$$\sum_{i=0}^N A^i = \frac{A^{N+1} - 1}{A - 1}$$

在第二个公式中, 如果  $0 < A < 1$ , 则

$$\sum_{i=0}^N A^i \leq \frac{1}{1-A}$$

当  $N$  趋于  $\infty$  时, 该和趋向于  $1/(1-A)$ 。这些公式是“几何级数”公式。

可以用下面的方法推导关于  $\sum_{i=0}^{\infty} A^i$  ( $0 < A < 1$ ) 的公式。令  $S$  是其和, 于是

$$S = 1 + A + A^2 + A^3 + A^4 + A^5 + \dots$$

于是

$$AS = A + A^2 + A^3 + A^4 + A^5 + \dots$$

如果将这两个方程相减 (这种运算只允许对收敛级数进行), 则等号右边1以外的所有项相消, 结果得到:

$$S - AS = 1$$

这就是说

$$S = \frac{1}{1-A}$$

可以用相同的方法计算  $\sum_{i=1}^{\infty} i/2^i$ , 这是一个经常出现的和。把它写成

$$S = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + \dots$$

两边乘以2得到

$$2S = 1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \frac{5}{2^4} + \frac{6}{2^5} + \dots$$

将这两个方程相减得到

$$S = 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \dots$$

因此,  $S = 2$ 。

4



经常用来分析的另一种级数是算术级数。任何这样的级数都可以用基本公式求其值。

$$\sum_{i=1}^N i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

例如，为求和  $2 + 5 + 8 + \cdots + (3k - 1)$ ，将其改写为  $3(1 + 2 + 3 + \cdots + k) - (1 + 1 + 1 + \cdots + 1)$ ，显然，这就是  $3k(k+1)/2 - k$ 。另一种记忆方法则是将第一项与最后一项相加（和为  $3k+1$ ），第二项与倒数第二项相加（和也是  $3k+1$ ），依次类推。由于有  $k/2$  个这样的数对，因此总和就是  $k(3k+1)/2$ ，这与前面的答案相同。

下面介绍的两个公式就没有那么常见了。

$$\sum_{i=1}^N i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

$$\sum_{i=1}^N i^k \approx \frac{N^{k+1}}{|k+1|} \quad k \neq -1$$

当  $k = -1$  时，后一个公式不成立。此时需要下面的公式，这个公式在计算机科学中远比在其他数学科目中用得更多。数  $H_N$  叫作调和数，其和叫作调和和。下面近似式中的误差趋向于  $\gamma \approx 0.577\,215\,66$ ，称为**欧拉常数**（Euler's constant）。

$$H_N = \sum_{i=1}^N \frac{1}{i} \approx \log_e N$$

下面两个公式只不过是一般的代数运算：

$$\sum_{i=1}^N f(N) = Nf(N)$$

$$\sum_{i=n_0}^N f(i) = \sum_{i=1}^N f(i) - \sum_{i=1}^{n_0-1} f(i)$$

#### 1.2.4 模运算

如果  $N$  整除  $A-B$ ，那么就说  $A$  与  $B$  模  $N$  同余（congruent），记为  $A \equiv B \pmod{N}$ 。直观地看，这意味着无论  $A$  或  $B$  被  $N$  除，所得余数都是相同的。因此， $81 \equiv 61 \equiv 1 \pmod{10}$ 。与等号一样，若  $A \equiv B \pmod{N}$ ，则  $A+C \equiv B+C \pmod{N}$  以及  $AD \equiv BD \pmod{N}$ 。

有许多定理适用于模运算，其中有些需要用数论来特别证明。本书将尽量少用模运算，前面

5

#### 1.2.5 证明方法

证明数据结构分析中的结论的两种最常见的方法是归纳法证明和反证法证明（偶尔也不得已用唬人法证明<sup>1</sup>，这只有教授们才用）。证明定理不成立的最好方法是举出一个反例。

##### 1. 归纳法证明

由归纳法进行的证明有两个标准的部分。第一步是证明基准情形（base case），就是确定定理对于某个（某些）小的（通常是退化的）值的正确性；这一步总是很简单的。接着，进行**归纳假设**（inductive hypothesis）。一般说来，这意味着假设定理对直到某个有限数  $k$  的所有情况都是成立的。然后使用这个假设证明定理对下一个值（通常是  $k+1$ ）也是成立的。至此定理得证（在  $k$  是有限的情形下）。

1. 就是用一句高深莫测的“显然”来代替证明。——译者注

作为一个例子, 证明斐波那契数, 就是  $F_0 = 1, F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, \dots, F_i = F_{i-1} + F_{i-2}$ , 满足对  $i \geq 1$ , 有  $F_i < (5/3)^i$  (有些定义规定  $F_0 = 0$ , 这只是将该级数平移一项)。为了证明这个不等式, 首先验证定理对平凡情形成立。容易验证  $F_1 = 1 < 5/3$  及  $F_2 = 2 < 25/9$ ; 这就证明了基准情形。假设定理对于  $i = 1, 2, \dots, k$  成立; 这就是归纳假设。为了证明定理, 需要证明  $F_{k+1} < (5/3)^{k+1}$ 。根据定义有

$$F_{k+1} = F_k + F_{k-1}$$

将归纳假设用于等号右边, 得到

$$\begin{aligned} F_{k+1} &< (5/3)^k + (5/3)^{k-1} \\ &< (3/5)(5/3)^{k+1} + (3/5)^2(5/3)^{k+1} \\ &< (3/5)(5/3)^{k+1} + (9/25)(5/3)^{k+1} \end{aligned}$$

化简为

$$\begin{aligned} F_{k+1} &< (3/5 + 9/25)(5/3)^{k+1} \\ &< (24/25)(5/3)^{k+1} \\ &< (5/3)^{k+1} \end{aligned}$$

定理得证。

作为第二个例子, 证明下面的定理。

**定理1.3** 如果  $N \geq 1$ , 则  $\sum_{i=1}^N i^2 = \frac{N(N+1)(2N+1)}{6}$ 。

**证明** 用数学归纳法证明。对于基准情形, 容易看到, 当  $N=1$  时定理成立。作为归纳假设, 假设定理对  $1 \leq k \leq N$  成立。将在该假设下证明定理对于  $N+1$  也是成立的。首先有

$$\sum_{i=1}^{N+1} i^2 = \sum_{i=1}^N i^2 + (N+1)^2$$

然后应用归纳假设得到

$$\begin{aligned} \sum_{i=1}^{N+1} i^2 &= \frac{N(N+1)(2N+1)}{6} + (N+1)^2 \\ &= (N+1) \left[ \frac{N(2N+1)}{6} + (N+1) \right] \\ &= (N+1) \frac{2N^2 + 7N + 6}{6} \\ &= \frac{(N+1)(N+2)(2N+3)}{6} \end{aligned}$$

因此

$$\sum_{i=1}^{N+1} i^2 = \frac{(N+1)[(N+1)+1][2(N+1)+1]}{6}$$

定理得证。 ■

## 2. 通过反例证明

公式  $F_k \leq k^2$  不成立。最容易的证明方法就是计算  $F_{11} = 144 > 11^2$ 。

## 3. 反证法证明

反证法证明是通过假设定理不成立, 然后证明该假设导致某个已知的性质不成立, 从而证明原假设是错误的。一个经典的例子是证明存在无穷多个素数。为了证明这个结论, 假设定理不成立。于是, 存在某个最大的素数  $P_k$ 。令  $P_1, P_2, \dots, P_k$  是依序排列的所有素数, 考虑



$$N = P_1 P_2 P_3 \cdots P_k + 1$$

显然， $N$ 是比 $P_k$ 大的数，根据假设 $N$ 不是素数。可是， $P_1, P_2, \dots, P_k$ 都不能整除 $N$ ，因为除得的结果总有余数1。这就产生一个矛盾，因为每一个整数，或者是素数，或者是素数的乘积。因此， $P_k$ 是最大素数的原假设是不成立的，这正意味着定理成立。

### 1.3 递归的简单介绍

人们熟悉的大多数数学函数是由一个简单公式描述的。例如，可以利用公式

$$C = 5(F - 32)/9$$

把华氏温度转换成摄氏温度。有了这个公式，写一个C++函数就太简单了。除去程序中的声明和大括号外，一行公式翻译成一行C++程序。

7 有时候用非标准的形式来定义数学函数。例如，可以在非负整数集上定义一个函数  $f$ ，它满足  $f(0) = 0$  且  $f(x) = 2f(x-1) + x^2$ 。从这个定义可以看出  $f(1) = 1, f(2) = 6, f(3) = 21, f(4) = 58$ 。当一个函数用自身来定义时就称为是递归 (recursive) 的。C++ 允许函数是递归的。<sup>1</sup>但必须要记住，C++ 所做的仅仅是试图遵循递归思想。不是所有的数学递归函数都能被有效地 (或正确地) 用 C++ 的递归模拟来实现。要点在于，递归函数  $f$  应该像非递归函数一样只用几行就能表示出来。图1-2给出了函数  $f$  的递归实现。

```

1 int f( int x )
2 {
3     if( x == 0 )
4         return 0;
5     else
6         return 2 * f( x - 1 ) + x * x;
7 }
```

图1-2 一个递归函数

第3行和第4行处理**基准情况** (base case)，即此时函数的值可以直接算出而不用求助递归。正如在没有  $f(0) = 0$  这个事实的前提下，声称  $f(x) = 2f(x-1) + x^2$ ，在数学上没有意义一样，C++ 的递归方法若无基准情况也是毫无意义的。第6行执行的是递归调用。

关于递归，有几个重要而可能混淆的概念。一个常见问题是：它就是循环逻辑吗？答案是：虽然用一个函数本身来定义这个函数，但是并没有用一个函数实例本身来定义该特定实例。换句话说，通过使用  $f(5)$  来得到  $f(5)$  的值才是循环的。通过使用  $f(4)$  得到  $f(5)$  的值就不是循环的，当然，除非  $f(4)$  的求值最终又要计算  $f(5)$ 。两个最重要的问题恐怕就是如何和为什么的问题了。如何和为什么的问题将在第3章正式讨论。这里，我们只给出一个不完整的描述。

其实，处理递归调用与处理任何其他调用并没有什么不同。如果以参数值4调用函数  $f$ ，那么程序的第6行要求计算  $2f(3) + 4 \times 4$ 。这样，就要执行一次计算  $f(3)$  的调用，而这又导致计算  $2f(2) + 3 \times 3$ 。因此，又要执行另一个计算  $f(2)$  的调用，而这意味着必须求出  $2f(1) + 2 \times 2$  的值。为此，通过计算  $2f(0) + 1 \times 1$  而得到  $f(1)$ 。此时， $f(0)$  必须被赋值。由于这属于基准情况，因此事先知道  $f(0) = 0$ 。从而  $f(1)$  的计算得以完成，其结果为1。然后， $f(2)$ 、 $f(3)$  以及最后  $f(4)$  都能确定下来。跟踪挂起的函数调用（这些调用已经开始但是正等待着一次递归调用的完成）以及跟踪这些调用的变量、所需的记录工作都由计算机自动完成。然而，要点在于，递归调用将一直进行到基准情形出现为止。

1. 对于数值计算，使用递归通常不是个好主意。这里这么做只是为了解释基本论点。

例如，计算 $f(-1)$ 的值将导致调用 $f(-2)$ 、 $f(-3)$ 等。由于这样永远不能达到基准情形，因此程序不能算出答案。偶尔还发生更加微妙的错误，如图1-3所示的示例。图1-3中程序的这种错误是第6行上的 $\text{bad}(1)$ 被定义为 $\text{bad}(1)$ 。显然，实际上 $\text{bad}(1)$ 究竟是多少，这个定义给不出任何线索。因此，计算机将会反复调用 $\text{bad}(1)$ 以期解出它的值。最后，计算机簿记系统将耗完内存空间，程序非正常终止。通常，我们会说该函数对一个特殊情形无效，而对其他情形是正确的。但此处这么说则不正确，因为 $\text{bad}(2)$ 调用 $\text{bad}(1)$ 。因此， $\text{bad}(2)$ 也不能求出值来。不仅如此， $\text{bad}(3)$ 、 $\text{bad}(4)$ 和 $\text{bad}(5)$ 都要调用 $\text{bad}(2)$ ， $\text{bad}(2)$ 算不出值，它们的值也就不能求出。事实上，除了0外，这个程序对任何非负的 $n$ 都无效。对于递归程序，不存在“特殊情形”。

```

1 int bad( int n )
2 {
3     if( n == 0 )
4         return 0;
5     else
6         return bad( n / 3 + 1 ) + n - 1;
7 }

```

图1-3 无终止递归函数

上面的讨论引出递归的前两个基本法则：

- (1) 基准情形 (base cases)。必须总有某些基准的情形，它们不用递归就能求解。
- (2) 不断推进 (making progress)。对于那些要被递归求解的情形，递归调用必须总能够朝着一个基准情形推进。

本书中将用递归解决一些问题。作为非数学应用的一个例子，考虑一本大词典。词典中的词都是用其他的词定义的。当我们查一个单词的时候，不是总能理解对该词的解释，于是我们不得不再查解释中的某些词。同样，对这些词中的某些词我们又不理解，因此还要继续这种搜索。因为词典是有限的，所以实际上：(1)或者我们最终查到一处，明白了此处解释中所有的单词（从而理解这里的解释，并按照查找的路径回头理解其余的解释）；(2)或者我们发现这些解释形成一个循环，无法理解其中的意思，也许在解释中需要我们理解的某个单词不在这本词典里。

理解这些单词的递归策略如下：如果我们知道一个单词的含义，那么就算成功；否则，就在词典里查找这个单词。如果我们理解对该词解释中的所有单词，那么也算成功；否则，通过递归地查找一些我们不认识的单词来弄明白对该单词解释的含义。如果词典编纂得完美无暇，那么这个过程就能够终止；如果其中一个单词没有查到或是形成循环定义（解释），那么这个过程则无限循环。

9

### 1. 打印输出整数

设有一个正整数 $n$ 并希望把它打印出来。我们的例程的名字为 $\text{printOut}(n)$ 。假设仅有的可用I/O例程将只处理单个数字并将其输出到终端，我们给这种例程命名为 $\text{printDigit}$ ；例如， $\text{printDigit}(4)$ 将输出一个4到终端。

递归为该问题提供一个非常简洁的解。为打印76234，需要首先打印出7623，然后再打印出4。第二步用语句 $\text{printDigit}(n\%10)$ 很容易完成，但是第一步却不比原问题简单多少。它实际上是同一个问题，因此可以用语句 $\text{printOut}(n/10)$ 递归地解决它。

这告诉我们如何去解决一般的问题，不过我们仍然需要确认程序不是无限循环的。由于尚未定义一个基准情况，因此很显然，我们仍然还有些事情要做。如果 $0 \leq n < 10$ ，那么基准情形就是 $\text{printDigit}(n)$ 。现在， $\text{printOut}(n)$ 已对每一个从0到9的正整数定义，而更大的正整数则用较小的正整数定义。因此，不存在循环。整个函数如图1-4所示。



```

1 void printOut( int n ) // Print nonnegative n
2 {
3     if( n >= 10 )
4         printOut( n / 10 );
5     printDigit( n % 10 );
6 }

```

图1-4 打印整数的递归例程

我们没有努力去高效地做这件事。这里是避免使用mod例程（其耗费很大）的，因为 $n \% 10 = n - \lfloor n/10 \rfloor \times 10$ 。<sup>1</sup>

## 2. 递归和归纳

下面将使用归纳法对上述递归的数字打印程序进行较为严格的证明。

**定理1.4** 对于 $n \geq 0$ ，递归的数字打印算法是正确的。

**证明（通过对 $n$ 所含数字的个数的归纳法证明之）** 首先，如果 $n$ 只有一位数字，那么程序显然是正确的，因为它只是调用一次printDigit。然后，设printOut对所有 $k$ 位或位数更少的数均能正常工作。 $k+1$ 位的数可以通过其前 $k$ 位数字后跟一位最低位数字来表示。但是前 $k$ 位数字形成的数恰好是 $\lfloor n/10 \rfloor$ ，由归纳假设它能够被正确地打印出来，而最后的一位数字是 $n \bmod 10$ ，因此

10 该程序能够正确打印出任意 $k+1$ 位的数。于是，根据归纳法，所有的数都能被正确地打印出来。■

这个证明看起来可能有些奇怪，它实际上相当于是算法描述。证明阐述的是在设计递归程序时，同一问题的所有较小实例均可以假设运行正确，递归程序只需要把这些较小问题的解（它们通过递归奇迹般地得到）结合起来而形成当前问题的解。其数学根据则是归纳法证明。于是引出递归的第3个法则：

(3) 设计法则（design rule）。假设所有的递归调用都能运行。

这是一条重要的法则，因为它意味着，当设计递归程序时一般没有必要知道簿记管理的细节，不必试图追踪大量的递归调用。追踪实际的递归调用序列常常是非常困难的。当然，在许多情况下，这正体现了使用递归的好处，因为计算机能够算出复杂的细节。

递归的主要问题是隐含的簿记开销。虽然这些开销几乎总是合理的（因为递归程序不仅简化了算法设计而且有助于给出更加简洁的代码），但是递归绝不应该作为简单for循环的代替物。3.6节将更仔细地讨论递归涉及的系统开销。

当编写递归例程的时候，关键是要牢记递归的四条基本法则：

(1) 基准情形。必须总有某些基准情形不用递归就能求解。

(2) 不断推进。对于那些需要递归求解的情形，递归调用必须总能够朝着基准情形的方向推进。

(3) 设计法则。假设所有的递归调用都能运行。

(4) 合成效益法则（compound interest rule）。在求解一个问题的同一实例时，切勿在不同的递归调用中做重复性的工作。

第4条法则（连同它的名称一起）将在后面的小节中给予证明。使用递归计算诸如斐波那契数之类简单数学函数的值一般来说不是一个好主意，其原因正是根据第4条法则。只要在头脑中记住这些法则，递归程序设计就应该是简单明了的。

1.  $\lfloor x \rfloor$ 是小于或等于 $x$ 的最大整数。

## 1.4 C++类

本书中提供了许多数据结构。所有的这些数据结构都是用来存储数据（通常是相同类型项的集合）的对象，并且提供处理这些集合的函数。在C++（或者其他编程语言）中，这通过使用类完成。本节讨论C++类。

11

### 1.4.1 基本class语法

在C++中类由**成员**（member）构成。成员可以是数据，也可以是函数，其中函数称为**成员函数**（member function）。类中的每一个实例都是一个对象。每一个对象包含类中指定的数据成员（除非这些数据成员是static，否则这是一个可以暂时安全忽略的细节）。成员函数作用于对象，通常被称为方法（method）。

图1-5是IntCell类的一个例子。在IntCell类中，IntCell的每一个实例（IntCell对象）都包含一个称为storedValue的数据成员。这个类中的其他部分是方法。在这个例子中共有4个方法，其中的2个方法是read和write，另外的2个是称为构造函数的特殊方法。下面论述某些关键特性。

```

1  /**
2   * A class for simulating an integer memory cell.
3   */
4  class IntCell
5  {
6      public:
7          /**
8           * Construct the IntCell.
9           * Initial value is 0.
10          */
11         IntCell( )
12             { storedValue = 0; }
13
14         /**
15          * Construct the IntCell.
16          * Initial value is initialValue.
17          */
18         IntCell( int initialValue )
19             { storedValue = initialValue; }
20
21         /**
22          * Return the stored value.
23          */
24         int read( )
25             { return storedValue; }
26
27         /**
28          * Change the stored value to x.
29          */
30         void write( int x )
31             { storedValue = x; }
32
33     private:
34         int storedValue;
35 };

```

图1-5 IntCell类的完整声明



首先，讨论两个标号public和private。这些标号声明类成员的可见性。在这个例子中除了storedValue是private的以外，其他的类成员都是public的。public的类成员可以被任何类中的任何方法访问。private的类成员仅可以被它所在类的方法访问。一般地，数据成员声明为private，这样可以禁止对该类内部细节的访问，而作为一般用途的方法则定义为public。这被称为**信息隐藏**（information hiding）。通过使用private数据成员，可以改变对象的内部代码而不影响程序里其他使用到这个对象的部分。这是因为对象的访问是通过public成员函数实现的，而该成员函数的可见性并没有改变。类的使用者并不需要知道类实现的内部细节。在许多情况下，对内部信息的访问会导致错误。例如，对于一个使用月、日和年存储日期的类，可以通过设置月、日和年的属性为private来禁止外部对这些数据成员的非法修改，如改成Feb 29, 2001。无论如何，内部使用的类都可以设成private的。在类中，所有的数据成员的默认属性都为private，因此，初始化public是不可选的。

其次，讨论两个**构造函数**。构造函数是描述如何构建类的实例的方法。如果没有显式定义的构造函数，那么可以自动生成使用编程语言的默认值来初始化数据成员的构造函数。IntCell类定义了两个构造函数。如果不指定参数，就调用第一个构造函数。如果提供int型参数，则调用第二个方法，并使用这个int参数来初始化storedValue成员。

### 1.4.2 特别的构造函数语法与访问函数

虽然类可以如上面描述的那样工作，还有一些特别的语法可以用来生成更好的代码。图1-6显示了四个改进（为了更简洁，这里忽略了注释）。这些区别如下。

```

1  /**
2   * A class for simulating an integer memory cell.
3   */
4  class IntCell
5  {
6  public:
7      explicit IntCell( int initialValue = 0 )
8          : storedValue( initialValue ) { }
9      int read( ) const
10         { return storedValue; }
11      void write( int x )
12         { storedValue = x; }
13
14 private:
15     int storedValue;
16 };

```

图1-6 改进的IntCell类

#### 1. 默认参数

IntCell构造函数阐述了**默认参数**（default parameter）。相应地，定义了两个IntCell 构造函数。一个构造函数接受initialValue，另一个是零参数构造函数。后者是隐含的，因为在单参数构造函数中initialValue是可选的。默认值0意味着，如果没有确定的参数，那么就使用0。

12 默认参数可以在任何函数中使用，但是最普遍的情况是用在构造函数中。

#### 2. 初始化列表

IntCell构造函数在其代码体之前使用**初始化列表**（initializer list）（图1-6，第8行）。初始化列表用来直接初始化数据成员。在图1-6中，这几乎没有什么区别，但是在数据成员是具有复杂初始化过程的类类型的时候，使用初始化列表代替代码体中的赋值语句可以节省很多时间。在某

些情况下，这是很必要的。例如，如果一个数据成员是const的（意味着在对象被构造后就不能再改变），那么，数据成员的值就只能在初始化列表里进行初始化。另外，如果一个数据成员是不具有零参数的构造函数的类类型，那么，该数据成员也必须在初始化列表里进行初始化。 13

### 3. explicit构造函数

IntCell构造函数是explicit的。所有的单参数的构造函数都必须是explicit的，以避免后台的类型转换。否则，一些宽松的规则将允许在没有显式类型转换操作的情况下进行类型转换。通常，这种不希望发生的行为会破坏代码的可读性，并导致难以发现的错误。考虑下面的例子：

```
IntCell obj;          //obj is an IntCell
obj = 37;             //Should not compile: type mismatch
```

上面的代码段构造了一个IntCell对象obj，并且进行了赋值。但是该赋值语句并不能工作，因为在赋值符号右侧并不是另一个IntCell对象。取而代之的应该是使用obj的write方法。然而，C++拥有宽松的规则。通常，单参数构造函数定义了一个**隐式类型转换**（implicit type conversion），该转换创建了一个临时对象，从而使赋值（或函数参数）变成兼容的。在本例中，编译器试图将

```
obj = 37;             //Should not compile: type mismatch
```

转换为

```
IntCell temporary = 37;
obj = temporary;
```

注意，临时对象的构造也可以通过使用单参数构造函数来实现。使用explicit意味着单参数构造函数不能用来创建隐式临时对象。这样一来，既然IntCell构造函数已经声明为explicit，那么编译器就能够正确地分析出这里有一个类型不匹配。 14

### 4. 常量成员函数

只进行检测但不改变其对象的状态的成员函数称为**访问函数**（accessor）。改变其对象的状态的成员函数称为**修改函数**（mutator）（因为该函数修改了该对象的状态）。例如，在典型的集合类中，isEmpty是访问函数，而makeEmpty是修改函数。

在C++中，每个成员函数都被标记为访问函数或修改函数。在设计阶段这是很重要的一步，不可以被简单地看成注释。事实上，这是重要的语义逻辑。例如，访问函数不能用于常量对象。默认情况下，所有的成员函数都是修改函数，要使成员函数成为访问函数必须在参数类型列表结尾的圆括号后加上关键字const。const可以具有很多不同的含义。函数声明在不同的情况下可以有三种方式使用const。只有跟在结尾圆括号后的const才表示一个访问函数。其他的两种用法在1.5.2节和1.5.3节中介绍。

在IntCell类中，read很明显是一个访问函数，它并不改变IntCell的状态。这样，第9行就是一个常量成员函数。如果成员函数标记为访问函数但在实现中又去改变数据成员的值，那么就会出现编译错误。<sup>1</sup>

#### 1.4.3 接口与实现的分离

图1-6中的类包含所有的正确句法结构。然而在C++中，将类的接口与其实现分离是很常见的。接口列出了类及其成员（数据和函数），而实现提供了函数的具体实现。

图1-7给出了类IntCell的接口，图1-8给出了其实现，而图1-9给出了一个使用IntCell的

1. 可以将数据成员标记为mutable，指示常量不适用于它们。

main例程。一些要点如下。

### 1. 预处理命令

接口通常都放在以.h结尾的文件中。需要接口信息的源代码必须#include接口文件。本例既是实现文件又是包含main的文件。偶尔，一个复杂的项目中有包含其他文件的文件，这样在编译一个文件时就存在一个接口被读两次的危险，这是非法的。为避免这种情况，每个头文件在读类接口时都定义一个预处理器来定义一个符号，如图1-7的前两行所示。符号名IntCell\_H不应该再出现在其他文件中，通常该符号都是文件名。接口文件的第一行检测该符号是否是未定义的。如果答案是肯定的，就接着处理文件，否则就不处理文件（跳到#endif），因为该文件已知是读过的了。

```

1  #ifndef IntCell_H
2  #define IntCell_H
3
4  /**
5   * A class for simulating an integer memory cell.
6   */
7  class IntCell
8  {
9      public:
10     explicit IntCell( int initialValue = 0 );
11     int read( ) const;
12     void write( int x );
13
14     private:
15     int storedValue;
16 };
17
18 #endif

```

图1-7 文件IntCell.h中的IntCell类接口

### 2. 作用域运算符

实现文件通常都是以.cpp、.cc或者.c结尾的，其中的成员函数都必须声明为类的一部分。否则，函数就会被认为是全局的（导致无数的错误）。语法是ClassName::member。::称为作用域运算符。

### 3. 签名必须精确匹配

实现的成员函数的签名必须与类接口中列出的签名精确匹配。回想刚才的例子，无论成员函数是访问函数（通过const在末尾声明）还是修改函数，都是签名的一部分。因此，如果在图1-7和图1-8中省略任何一个read签名后的const都会导致错误。注意，默认参数仅在接口中被定义，在实现中则被忽略。

### 4. 如基本类型一样声明对象

在C++中，对象如同基本类型一样声明。于是，下面对IntCell对象的声明是合法的：

```

IntCell obj1;           // Zero parameter constructor
IntCell obj2(12);       // One parameter constructor

```

而下面的声明就是错误的：

```

IntCell obj3 = 37;      // Constructor is explicit
IntCell obj4();         // Function declaration

```

obj3的声明之所以非法，是因为单参数构造函数是explicit的；否则它就是对的了。（换句



话说，使用单参数构造函数的声明必须使用圆括号来赋初始值。) obj4的声明说明它是一个函数(在其他地方定义的)，该函数没有参数并且返回IntCell。

```

1  #include "IntCell.h"
2
3  /**
4   * Construct the IntCell with initialValue
5   */
6  IntCell::IntCell( int initialValue ) : storedValue( initialValue )
7  {
8  }
9
10 /**
11  * Return the stored value.
12  */
13 int IntCell::read( ) const
14 {
15     return storedValue;
16 }
17
18 /**
19  * Store x.
20  */
21 void IntCell::write( int x )
22 {
23     storedValue = x;
24 }

```

图1-8 文件IntCell.cpp中的IntCell类实现

```

1  #include <iostream>
2  #include "IntCell.h"
3  using namespace std;
4
5  int main( )
6  {
7      IntCell m;    // Or, IntCell m( 0 ); but not IntCell m( );
8
9      m.write( 5 );
10     cout << "Cell contents: " << m.read( ) << endl;
11
12     return 0;
13 }

```

图1-9 文件TestIntCell.cpp中使用IntCell的程序

#### 1.4.4 vector和string

C++标准定义了两个类vector和string。vector意在替代带来无穷麻烦的C++内置数组。使用C++内置数组的问题在于其行为与基本类对象不同。例如，内置数组不能使用=复制，也不能记忆其本身能存储多少项，并且其索引操作不能检查索引是否有效。内置字符串仅是字符数组，这并没怎么提高数组的功效。例如，==不能正确地比较两个内置字符串。

在STL中，vector和string类将数组和字符串作为基本类来处理。vector有确定的大小。两个string对象可以用==和<等进行比较。vector和string都可以用=进行复制。如果可能，应该避免使用C++内置数组和字符串。在第3章论述vector如何实现的部分，我们将讨论C++内置数组。

vector和string都很容易使用。图1-10的代码创建了一个存储100个平方值并且将这些数据

输出的vector。注意，这里的size是一个方法，返回vector的大小。第3章将要谈到vector的一个优良特性是其大小很容易修改。在许多情况下，初始的大小都是0，然后vector的大小按需要增长。

18

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main( )
6  {
7      vector<int> squares( 100 );
8
9      for( int i = 0; i < squares.size( ); i++ )
10         squares[ i ] = i * i;
11
12     for( int i = 0; i < squares.size( ); i++ )
13         cout << i << " " << squares[ i ] << endl;
14
15     return 0;
16 }

```

图1-10 使用vector类：存储100个平方值并将其输出

string也很容易使用并且具有所有比较两个字符串状态的关系运算符和相等运算符。如果字符串的值相等，str1==str2为true。string也具有length方法来返回字符串的长度。

## 1.5 C++细节

与其他任何语言一样，C++也有其自身的细节和语言特性。本节讨论其中的一部分。

### 1.5.1 指针

**指针变量**是用来存储其他对象的存储地址的变量。它是许多数据结构中使用的基本机制。例如，为了存储一系列项，可以用连续的数组来实现。但是，在连续的数组中间插入一项，需要通过移动许多项来完成。比存在数组中更好的方法是将每一项都存储在分离的不邻接的存储空间里，这些存储空间在程序运行的时候自动分配。每个对象都有一个链接来连到下一个对象。这个链接就是一个指针变量，因为它存储了另一个对象的存储地址。这就是第3章将要讨论的经典的链表。

为说明指针的操作，重写图1-9来动态分配IntCell。必须要强调的是，如果仅是为一个简单的IntCell类，是没有必要按这种方式写C++代码的。这里仅仅是为了简要地说明动态内存分配。在后续的内容里，我们将会遇到更复杂的类，从而看到这种技术是很有用而且很必要的。新版本的代码如图1-11所示。

```

1  int main( )
2  {
3      IntCell *m;
4
5      m = new IntCell( 0 );
6      m->write( 5 );
7      cout << "Cell contents: " << m->read( ) << endl;
8
9      delete m;
10     return 0;
11 }

```

图1-11 使用指针的IntCell程序（不是必须这样做的）

### 1. 声明

第3行是m的声明。\*说明m是一个指针变量，可以用来指向一个IntCell对象。m的**值**是它所指向的对象的地址。m在这时没有初始化。在C++中，在使用指针前并不对指针是否初始化进行检查（但是，有些供货商的编译器进行附加的检查）。使用未初始化的指针通常会破坏程序，因为这些指针会导致对不存在的存储地址的访问。一般来说，合并第3行和第5行或者初始化m为NULL指针都是好主意。

### 2. 动态对象创建

第5行动态创建了一个对象。在C++中，new返回指向新建对象的指针。在C++中有两种方式可以使用零参数构造函数来创建对象。下面的两种写法都是合法的：

```
m = new IntCell();    // OK
m = new IntCell;      // Preferred in this text
```

由于在1.4.3节中obj4说明的可能出现的函数声明的问题，一般采用第二种写法。

19

### 3. 垃圾收集及delete

在一些语言中，当一个对象不再引用时，就纳入自动垃圾收集的范围。程序员不需要再担心它。C++没有垃圾收集。当一个通过new来分配地址的对象不再引用时，就需要使用delete操作（通过指针）将其删除。否则，该对象所占的内存就不能释放（直到程序结束）。这被称为**内存泄漏**（memory leak）。很不幸的是，内存泄漏在许多C++程序中普遍存在。幸好内存泄漏在仔细编写的源代码中可以被自动释放。一个重要的规则就是能用**自动变量**的时候就不用new。在前面的程序中，IntCell不是用new来分派内存的，而是定义了一个局部变量。在这样的情况下，分配给IntCell的内存存在声明IntCell的函数返回时就自动被收回了。图1-11第9行给出了delete 操作符。

### 4. 指针的赋值和比较

在C++中指针变量的赋值和比较是基于指针变量的值，也就是说它所存储的地址。这样一来，如果两个指针变量指向同一个对象，那么它们就是相等的；如果指向不同的对象，那么，即使指针变量指向的对象本身是相等的，指针变量也是不等的。如果lhs和rhs是指针变量（兼容的类型），那么，lhs=rhs使lhs指向rhs所指向的同一个对象<sup>1</sup>。

### 5. 通过指针访问对象的成员

如果指针变量指向类类型的对象，那么该对象的（可见的）成员就可以通过->操作符进行访问。图1-11的第6和第7行给出了示例。

### 6. 其他指针运算

C++允许对指针进行各种特殊的运算，这些运算偶尔是很有用的。例如，<就可以使用。对于指针lhs和rhs，当lhs指向的对象的存储地址低于rhs所指向对象的存储地址时，lhs<rhs就为真。这样做通常并不是一个好主意。7.8节例举了一个不常见的相等运算。

20

一个重要的操作符是**取地址运算符**（&）。该操作符返回对象所在的内存地址，在1.5.5节讨论的别名测试的实现中有很重要的应用。

## 1.5.2 参数传递

包括C和Java在内的许多编程语言都是使用**按值调用**（call by value）来传递参数的，即将实参复制给形参。但是在C++中，参数有可能是大的复杂对象，导致复制的效率很低。另外，有时候

1. 在本书中，我们始终使用 lhs 和 rhs 来定义二元操作符的左手侧和右手侧。



又需要改变被传递的值。对此，C++有三种不同的方式来传递参数。有一个简单的规则可以确定具体采用哪种方式。

下面的函数声明阐述了这三种参数传递机制，该函数返回arr中前n个整数的平均值，并且如果n大于arr.size()或者小于1，就设定errorFlag为true：

```
double avg( const vector<int> & arr, int n, bool & errorFlag);
```

这里，arr是vector<int>类型的，使用**按常量引用调用**（call by constant reference）来传递。n是int类型的，通过**按值调用**来传递。errorFlag是bool类型的，使用**引址调用**（call by reference）来传递。参数传递机制的选用可以通过以下两步的判断来决定：

(1) 如果形参必须能够改变实参的值，那么就必须使用引址调用。

(2) 否则，实参的值不能被形参改变。如果参数类型是简单类型，使用按值调用。否则，参数类型是类类型的，一般按常量引用调用来传递<sup>1</sup>。

在avg的声明中，errorFlag参数是通过引址调用来传递的，所以errorFlag的新值返回到实参中。arr和n就不会被avg改变。arr按常量引用传递，因为它是类类型的，对其进行复制耗费巨大。n通过值来传递，因为它是基本类型，对其进行复制消耗很小。

参数传递选项总结如下：

(1) 按值调用适用于不被函数更改的小对象。

(2) 按常量引用调用适用于不被函数更改的大对象。

(3) 引址调用适用于所有可以被函数更改的对象。

21

### 1.5.3 返回值传递

对象的返回也可以是按值返回和按常量引用返回，偶尔也用到引址返回。多数情况下，不要使用引址返回。在1.7.2节的例子中，我们可以看到引址返回是很有用的，但很少这样做。

使用按值返回总是很安全的。但是，如果返回的对象是类类型的，更好的办法是使用按常量引用返回以节省复制的开销<sup>2</sup>。然而，这只在下面的情况下才可能：必须确保返回语句中的表达式在函数返回时依然有效。这在C++中是个非常棘手的问题，许多编译器都不能对错误的应用给出报警信息。

作为一个例子，看一下图1-12中的代码，其中包含两个几乎一样的用来在数组中寻找最大（按字母顺序）的string的函数。两者都试图按常量引用返回值。第一个版本findMAX是可用的——表达式a[MaxIndex]索引的vector是在findMax外部的，并且存在时间长于调用返回的时间。第二个版本是错的。maxValue是一个局部变量，当函数返回的时候就不复存在了。这样一来，没有进行复制之前就返回是不恰当的。如果编译器没有查出这个问题，那么返回值既可能是对的，也可能是错的。这会决于编译器释放maxValue所使用的内存的速度。这会使调试工作极为困难。

22

### 1.5.4 引用变量

引用变量和常量引用变量常用于参数传递。它们也可以用作局部变量或类的数据成员。在这些情况下，变量名就是它所引用的对象名的同义词（很像在引址调用中，许多形参名都是实参名的同义词）。作为局部变量，它们避免了复制的成本，因此在对含有类类型集合的数据结构进行

1. 然而，小的类类型（例如，存储单一内置类型的）可以不使用按常量引用调用而是使用按值调用。

2. 这里的常量（const）是指返回的对象自身以后不能修改。这不同于参数里的const和访问函数里的const。

排序时非常有用。这样，在许多情况下，客户端代码

```
string x = findMax(a);
...
cout << x << endl;
```

比下面的代码要好：

```
const string & x = findMax(a);
...
cout << x << endl;
```

```
1 const string & findMax( const vector<string> & arr )
2 {
3     int maxIndex = 0;
4
5     for( int i = 1; i < arr.size( ); i++ )
6         if( arr[ maxIndex ] < arr[ i ] )
7             maxIndex = i;
8
9     return arr[ maxIndex ];
10 }
11
12 const string & findMaxWrong( const vector<string> & arr )
13 {
14     string maxVal = arr[ 0 ];
15
16     for( int i = 1; i < arr.size( ); i++ )
17         if( maxVal < arr[ i ] )
18             maxVal = arr[ i ];
19
20     return maxVal;
21 }
```

图1-12 两个查找最大字符串的函数——仅第一个是对的

在第5章可以看到，第二种用法是为了重命名一个具有复杂表达式的对象而使用的局部引用变量。其代码与下面的代码相似：

```
list<T> & whichList = theLists[hash( x, theLists.size())];
if( find( whichList.begin(), whichList.end(), x ) != whichList.end() )
    return false;
else
    whichList.push_back(x);
```

使用引用变量后，复杂得多的表达式`theLists[hash(x,theLists.size())]`就不需要写（然后求值）4次了。

虽然本书中没有举例，但引用变量可以用作类数据成员。引用必须被初始化（为它们将要引用的对象）。

### 1.5.5 三大函数：析构函数、复制构造函数和operator=

在C++中，伴随类的是已经写好的三个特殊函数，它们是**析构函数**、**复制构造函数**和**operator=**。在许多情况下，都可以采用编译器提供的默认操作。有些时候却不行。

#### 1. 析构函数

当一个对象超出其作用域或执行delete时，就调用析构函数。通常，析构函数的唯一任务就是释放使用对象时所占有的所有资源。这其中包括为每一个相应的news调用delete，以及关

闭所有打开的文件等。默认操作是对每一个数据成员都使用析构函数。

## 2. 复制构造函数

有一种特殊的构造函数，用于构造新的对象，被初始化为相同类型对象的一个副本，这就是**复制构造函数**（copy constructor）。对任何对象，如IntCell对象，复制构造函数可以以如下实例进行调用：

- 声明的同时初始化，例如

```
IntCell B = C;
IntCell B( C );
```

而不是

```
B = C; //Assignment operator, discussed later
```

- 正如前面提及的，使用按值调用传递（而不是通过&或constant &）的对象无论如何都应该尽量少用。
- 通过值（而不是通过&或const &）返回对象。

第一种情况是最容易理解的，因为构造的对象是显式需要的。第二和第三种情况构造了用户不可见的临时对象。尽管如此，构造就是构造，在这两种情况下，我们还是将对象复制到了新创建的对象。

默认情况下，复制构造函数通过将复制构造函数依次应用到每一个数据成员来实现。对于简单数据类型（例如，int、double或指针）的数据成员，进行简单赋值就可以。在IntCell类中的数据成员storedValue就是这种情况。对于本身就是类对象的数据成员，每个数据成员的类的复制构造函数都依次作用于该数据成员。

## 3. operator=

当=应用于两个已经构造的对象时，就调用**复制赋值运算符**operator=。lhs=rhs试图复制rhs的状态至lhs。默认情况下，operator=通过将其依次应用于每个数据成员来实现。

## 4. 默认值带来的问题

观察IntCell类，可以看到默认值非常适用，所以，我们不需要再做其他任何工作。这是常见的情况。如果一个类由基本类型的数据成员和默认值适用的对象组成，那么类的默认值通常也就都是适用的。这样其数据成员是int、double、vector<int>、string，甚至是vector<string>的类都接受默认值。

24 主要问题出现在其数据成员是指针的类。第3章将详细讨论这个问题及其解决方案。在这里，我们仅仅勾勒这个问题的轮廓。假设类仅包含一个指针数据成员，并且这个指针指向一个动态分配地址的对象。默认的析构函数不对指针进行任何操作（一个好理由就是释放这个指针就必须删除自身）。而且，复制构造函数和operator=都不复制指针所指向的对象，而是简单地复制指针的值。这样一来，就得到了两个类实例，它们包含的指针都指向了同一个对象。这被称为是**浅复制**（shallow copy）。一般，我们期望得到的是对整个对象进行克隆的**深复制**（deep copy）。于是，当一个类含有的数据成员为指针并且深复制很重要的时候，一般的做法就是必须实现析构函数、operator=和复制构造函数。

对IntCell，这些运算的签名是

```
~IntCell(); //destructor
IntCell( const IntCell & rhs); // copy constructor
const IntCell & operator=(const IntCell & rhs);
```

虽然对于IntCell类来说默认值是可用的，如图1-13所示，我们还是写出了其实现。对析构函数



来说，程序体执行完毕后，就会为数据成员自动调用析构函数。因此，默认值就是空程序体。对复制构造函数来讲，默认值就是跟随程序体执行的复制构造函数的初始化列表。注意，如果在初始化列表里什么都没有，没有执行复制的话，那么，每个成员函数就按默认值（零参数）初始化。

```

1  IntCell::~IntCell( )
2  {
3      // Does nothing, since IntCell contains only an int data
4      // member. If IntCell contained any class objects, their
5      // destructors would be called.
6  }
7
8  IntCell::IntCell( const IntCell & rhs ) : storedValue( rhs.storedValue )
9  {
10 }
11
12 const IntCell & IntCell::operator=( const IntCell & rhs )
13 {
14     if( this != &rhs )    // Standard alias test
15         storedValue = rhs.storedValue;
16     return *this;
17 }

```

图1-13 三大函数的默认值

operator=是我们最感兴趣的。第14行是一个别名测试，以确保我们没有复制自身。假设没有复制自身，对每一个数据成员应用operator=（在第15行）。然后在第16行返回一个对当前对象的引用。于是赋值可以链状进行，如a=b=c。

在我们所写的例程中，如果默认值有意义，就总是接受默认值。然而，如果默认值没有意义，就必须实现析构函数、operator=和复制构造函数。当默认值不能正常工作时，复制构造函数一般来说都可以通过模拟正常的构造，然后再调用operator=来实现。另一个常用的办法是给出一个合理的复制构造函数的实现，然后将其放在private部分里以屏蔽按值调用。

25

### 5. 当默认值不可用时

最常见的默认值不可用的情况是，数据成员是指针类型的，并且被指对象通过某些对象成员函数（例如构造函数）来分配地址。举例说明，假设IntCell是通过动态分配一个int来实现的，如图1-14所示。简单起见，此处不对接口和实现进行分离。

```

1  class IntCell
2  {
3  public:
4      explicit IntCell( int initialValue = 0 )
5          { storedValue = new int( initialValue ); }
6
7      int read( ) const
8          { return *storedValue; }
9      void write( int x )
10         { *storedValue = x; }
11 private:
12     int *storedValue;
13 };

```

图1-14 数据成员是指针，默认值不适用

在图1-15中暴露出了大量的问题。首先，虽然逻辑上只有a为4，但实际上输出了三个4。问题就在于默认的operator=和复制构造函数都是复制指针storedValue。这样一来，

a.storedValue、b.storedValue和c.storedValue都指向了同一个int变量。这些复制都是浅复制：指针被复制而不是指针所指的对象被复制。其次，另一个不明显的问题是内存泄漏。a的构造函数初始化的int变量依然存在，其内存需要释放。c的构造函数初始化的int变量不再有任何指针变量来引用。这也需要将其内存进行释放，但是已经没有指针存储该地址了。

```

1 int f( )
2 {
3     IntCell a( 2 );
4     IntCell b = a;
5     IntCell c;
6
7     c = b;
8     a.write( 4 );
9     cout << a.read( ) << endl << b.read( ) << endl << c.read( ) << endl;
10    return 0;
11 }

```

图1-15 暴露出图1-14中问题的简单函数

我们应用三大函数来解决这些问题。图1-16给出了实现的结果（接口和实现分离）。一般来说，如果析构函数是释放内存所必需的，那么复制赋值和复制构造函数的默认值就不适用。

如果类所包含的成员函数不能复制自身，那么operator=的默认值就不可用。后续的内容里我们会看到一些这方面的例子。

### 1.5.6 C风格的数组和字符串

C++语言提供内置的C风格的数组类型。要定义一个有10个整数的数组arr1，可以这样写：

26

```
int arr1[10];
```

arr1事实上是一个指向足够存储10个int的内存的指针，而不是基本类数组类型的。对数组使用=的结果是复制两个指针的值，而不是整个数组。而且对于上面的声明这样做也是非法的，因为arr1是常量指针。当arr1传递给一个函数时，只有指针的值被传递，而数组的大小信息却丢失了。因此，数组的大小必须作为一个附加参数来传递。由于数组的大小未知，也就没有关于索引范围的检查。

在上面的定义中，在编译的时候数组的大小必须是已知的。10不可以用变量代替。如果数组的大小未知，就必须显式声明一个指针，并且用new[]来分配内存。例如，

```
int *arr2 = new int[ n ];
```

现在arr2的作用就和arr1一样了，只不过它不是常量指针。这样，该指针就可以用来指向一大块的内存。然而，因为内存是动态分配的，在某些地方就必须通过delete[]来释放：

```
delete[] arr2;
```

否则的话，就会导致内存泄漏，并且，如果数组很大的话，泄漏会很严重。

内置的C风格的字符串是当作字符数组来实现的。特殊的终止符'\0'用以标识字符串逻辑上的结束，以避免传递字符串的长度值。字符串可以通过strcpy来复制，也可以通过strcmp来比较。字符串的长度可以通过strlen来确定。每个字符都可以通过数组索引操作来访问。这些字符串有数组所具有的所有问题，包括困难的内存管理问题。譬如当字符串进行复制时，总是假设目标数组的空间是足够大的。如果空间不够大的话，常会因为没有存储字符串终止符的空间而导致调试的极大困难。

```

1 class IntCell
2 {
3     public:
4         explicit IntCell( int initialValue = 0 );
5
6         IntCell( const IntCell & rhs );
7         ~IntCell( );
8         const IntCell & operator=( const IntCell & rhs );
9
10        int read( ) const;
11        void write( int x );
12    private:
13        int *storedValue;
14 };
15
16 IntCell::IntCell( int initialValue )
17 {
18     storedValue = new int( initialValue );
19 }
20
21 IntCell::IntCell( const IntCell & rhs )
22 {
23     storedValue = new int( *rhs.storedValue );
24 }
25
26 IntCell::~~IntCell( )
27 {
28     delete storedValue;
29 }
30
31 const IntCell & IntCell::operator=( const IntCell & rhs )
32 {
33     if( this != &rhs )
34         *storedValue = *rhs.storedValue;
35     return *this;
36 }
37
38 int IntCell::read( ) const
39 {
40     return *storedValue;
41 }
42
43 void IntCell::write( int x )
44 {
45     *storedValue = x;
46 }

```

图1-16 数据成员是指针，需要使用三大函数

标准的vector类和string类在实现的时候隐藏了内置的C风格的数组和指针的操作。第3章讨论vector类的实现问题。使用vector和string几乎总是较好的选择，但是，当使用同时为C和C++设计的库例程时也可能必须用到C风格的数组和字符串。偶尔（但是极少）为优化程序的运行速度，也会在代码中用到一小部分这些C风格的数组和字符串。

27  
 28

## 1.6 模板

考虑一下在数组中寻找最大项的问题。简单的算法是顺序扫描，一次检测一项，跟踪最大值。



与许多算法的典型特征一样，顺序扫描算法是类型无关的。所谓类型无关，就是说这种算法的逻辑与存储在数组中的项的类型无关。相同的逻辑可以适用于整数、浮点数或者具有可比性的任何类型。

本书描述的算法和数据结构都是类型无关的。当编写C++代码的类型无关的算法或数据结构时，我们更愿意只写一次，而不是为不同的类型都重写一次。

在本节中，我们将要论述在C++中如何用**模板**（template）来写类型无关的算法（也称为泛型算法）。我们先讨论函数模板，然后是类模板。

### 1.6.1 函数模板

函数模板通常都很容易书写。**函数模板**（function template）不是真正的函数，而是一个用以产生函数的公式。图1-17显示的findMax函数模板与图1-12显示的string的例程完全相同。含有template声明的行显示Comparable是模板实参，该实参可以被任何类型代替来生成函数。例如，用vector<string>作为参数的findMax函数被调用时，就通过用string替换Comparable来生成一个新的函数。

```

1  /**
2   * Return the maximum item in array a.
3   * Assumes a.size( ) > 0.
4   * Comparable objects must provide operator< and operator=
5   */
6  template <typename Comparable>
7  const Comparable & findMax( const vector<Comparable> & a )
8  {
9      int maxIndex = 0;
10
11     for( int i = 1; i < a.size( ); i++ )
12         if( a[ maxIndex ] < a[ i ] )
13             maxIndex = i;
14     return a[ maxIndex ];
15 }
```

图1-17 findMax函数模板

如图1-18所示，函数模板可以应需要而自动扩展。要注意的是随着每种类型的扩展，都会生成附加代码。在大项目里，这被称为**代码膨胀**（code bloat）。还应该注意调用findMax(v4)会导致编译时出错。这是因为当IntCell替换了Comparable后，图1-17的第12行变成非法的：没有为IntCell定义<函数。因此，在使用任何模板前，习惯上都是先包含用以说明关于类模板实参的定义的注释。这也包含了关于需要哪种类型的构造函数的定义。

因为在决定参数传递和返回值传递时，模板实参可以使用任何类类型，所以，模板的实参应该定义为非基本数据类型。这也是采用常量引用的原因。

有很多处理函数模板的晦涩的规则，这并不稀奇。大多数的都是出现在函数模板不能提供完全匹配的而只是相近的参数（通过隐式类型转换）的情况下。必须有解决这种不确定问题的办法，其规则也是非常复杂的。注意，如果有一个非模板和一个模板都可以匹配的话，那么非模板具有优先权。还要注意到，如果有两个同样相似的匹配，那么代码就是非法的，并且编译器会报错。

### 1.6.2 类模板

在最简单的版本里，类模板与函数模板的运行情况相似。图1-19给出了MemoryCell模板。

MemoryCell与IntCell类相似，但是可以用于任何类型的Object，只要Object具有一个零参数构造函数、一个复制构造函数和一个复制赋值运算符。

```

1 int main( )
2 {
3     vector<int>    v1( 37 );
4     vector<double> v2( 40 );
5     vector<string> v3( 80 );
6     vector<IntCell> v4( 75 );
7
8     // Additional code to fill in the vectors not shown
9
10    cout << findMax( v1 ) << endl; // OK: Comparable = int
11    cout << findMax( v2 ) << endl; // OK: Comparable = double
12    cout << findMax( v3 ) << endl; // OK: Comparable = string
13    cout << findMax( v4 ) << endl; // Illegal; operator< undefined
14
15    return 0;
16 }
```

图1-18 使用findMax函数模板

```

1 /**
2  * A class for simulating a memory cell.
3  */
4 template <typename Object>
5 class MemoryCell
6 {
7     public:
8         explicit MemoryCell( const Object & initialValue = Object( ) )
9             : storedValue( initialValue ) { }
10         const Object & read( ) const
11             { return storedValue; }
12         void write( const Object & x )
13             { storedValue = x; }
14     private:
15         Object storedValue;
16 };
```

图1-19 未分离的MemoryCell类模板

要注意的是，Object是通过常量引用来传递的。还要注意到构造函数的默认参数不是0，因为0有可能不是有效的Object。取而代之，默认参数是通过零参数构造函数来构造Object的结果。

图1-20给出了MemoryCell怎样存储基本类型和类类型对象的例子。注意，MemoryCell不是类，而仅仅是类模板。MemoryCell<int>和MemoryCell<string>是真实的类。

```

1 int main( )
2 {
3     MemoryCell<int>    m1;
4     MemoryCell<string> m2( "hello" );
5
6     m1.write( 37 );
7     m2.write( m2.read( ) + "world" );
8     cout << m1.read( ) << endl << m2.read( ) << endl;
9
10    return 0;
11 }
```

图1-20 使用MemoryCell类模板的程序

31 如果将类模板作为一个单独的单元来实现，那么就很少有语法累赘。事实上，许多类模板就是用这种方式实现的。因为目前模板的分离编译在许多平台上都不能很好地运行。因此，在许多情况下，包括其实现的整个类都必须放在.h文件中。流行的STL的实现就是遵循这个策略。

在附录中讨论的另一个可选的办法是将类模板的接口和实现进行分离。这增加了额外的语法累赘，并且历史上曾经使编译器处理起来变得很困难。为避免额外的语法累赘，在本书中所提供的联机代码中，类模板的接口和实现都没有进行分离。在图中，接口看起来像是使用了分离的编译，而成员函数的实现却仿佛没有使用分离的编译一样。这使我们不必将精力集中在语法上。

### 1.6.3 Object、Comparable和例子

在本书中，我们总是将Object和Comparable作为泛型类型来使用。Object定义为含有一个零参数构造函数、一个operator=和一个复制构造函数。正如在findMax例子中推荐的，Comparable在operator<作用下具有可以提供全部数据的排序的附加功能<sup>1</sup>。

图1-21给出了一个实现时需要使用Comparable的类类型的例子，并且例举了**操作符重载**

```

1 class Employee
2 {
3     public:
4         void setValue( const string & n, double s )
5             { name = n; salary = s; }
6
7         const string & getName( ) const
8             { return name; }
9         void print( ostream & out ) const
10            { out << name << " (" << salary << ")"; }
11         bool operator< ( const Employee & rhs ) const
12            { return salary < rhs.salary; }
13
14         // Other general accessors and mutators, not shown
15     private:
16         string name;
17         double salary;
18 };
19
20 // Define an output operator for Employee
21 ostream & operator<< ( ostream & out, const Employee & rhs )
22 {
23     rhs.print( out );
24     return out;
25 }
26
27 int main( )
28 {
29     vector<Employee> v( 3 );
30
31     v[0].setValue( "George Bush", 400 000.00 );
32     v[1].setValue( "Bill Gates", 2 000 000 000.00 );
33     v[2].setValue( "Dr. Phil", 13 000 000.00 );
34     cout << findMax( v ) << endl;
35
36     return 0;
37 }

```

图1-21 Comparable可以是类类型，例如Employee

1. 第12章的部分数据结构使用operator==作为operator<的附加功能。注意，在全部数据排序中，如果a<b并且b<a都为假，那么a==b就为真。因此，使用operator==仅仅是为了方便起见。



(operator overloading)。操作符重载可以允许重新定义内置操作符的含义。Employee类包含name和salary, 并且在salary的基础上定义了operator<。更复杂的operator<也是可能的, 例如, 通过使用name成员函数来打破salary中的关联。Employee类也提供了零参数构造函数、operator=和复制构造函数(都是采用默认值)。这样Employee类在findMax中作为Comparable来使用就足够了。

为具有实用性, 每一个数据成员都必须是public的, 否则就必须提供附加的访问函数和修改函数。图1-21例举了setValue成员函数和为新类类型提供输出函数的广泛使用的惯用例程。这个惯用例程是提供一个命名为print的public成员函数。该函数使用ostream作为形参。该public成员函数可以被接受ostream和输出对象的全局非类函数operator<<调用<sup>1</sup>。

32  
33

### 1.6.4 函数对象

在1.6.1节, 我们论述了如何用函数模板来实现泛型算法。图1-17给出了一个用函数模板查找数组中最大项的例子。

然而, 模板有一个重要的限制: 只能用于定义了operator<函数的对象, 并且使用operator<作为所有比较的基础。在许多情况下, 这种方法并不可行。例如, 假定一个数据成员为姓名、工资、年龄和工作年限的Employee类含有operator<函数就很牵强。如图1-17的Employee类的例子所示, 即使含有了该函数, operator<函数也可能不是我们想要的。也许, 我们想要的不是最高工资而是最高工作年限。第二个例子, 假设我们想要查找数组中最大的字符串(按字母顺序), 由于默认的operator<不忽略大小写的区别, 就导致“ZEBRA”在“alligator”前面的情况。这可能也不是我们期望的结果。第三个例子就是由指向对象的指针所构成的数组(这在高级C++程序中是很平常的, 是称为继承的特性的应用, 在本书中没有应用该特性)。

这些问题的解决方案就是重写findMax, 使其可以像接受参数一样接受对象数组和比较函数, 并由该比较函数来决定两个对象中哪一个更大、哪一个更小。在实际效果上, 数组对象不再知道如何进行相互比较, 取而代之, 这些信息完全从数组对象中剥离出来。

注意, 对象同时包含数据和成员函数, 一个如传递参数一样传递函数的巧妙的办法是: 定义一个包含零个数据和一个成员函数的类, 然后传递这个类的实例。从效果上看就是, 通过将其放在对象中实现了函数的传递。该对象通常称为**函数对象**(function object)。

图1-22是函数对象思想的最简单实现。findMax获得第二个形参, 该形参为泛型类型。为使findMax模板可以无误地扩展, 泛型类型必须含有名为isLessThan的成员函数。该成员函数获得第一个泛型类型(Object)的两个形参, 并返回一个bool量。否则的话, 当编译器试图扩展模板的时候就会在第9行出错。在第26行, findMax被调用的时候传递了一个字符串数组和一个包含两个字符串参数的isLessThan方法。

C++的函数对象通过这种基本的思想来实现, 但有一些奇特的语法。首先, 不是使用函数名来调用函数, 而是使用操作符重载。不是使用函数isLessThan, 而是使用operator()。其次, 当调用operator()时, cmp.operator()(x,y)可以缩写为cmp(x,y)(换句话说, 这看起来就像是函数调用, 因此, operator()被称为**函数调用操作符**(function call operator))。结果, 参数

1. 这个惯例的一个替代方案就是用operator<<来直接实现print中的逻辑。因为operator<<不是类成员, 这就需要建立一个Employee类的friend函数。这涉及更多的C++语法。这样的替代方案有另一个附加的缺点, 就是在没有对全局模板函数提供friend声明的老编译器上不能正常工作。另一个缺点是不能在有继承性的更复杂的场合使用。这部分内容超出了本书的讨论范围。

```

1 // Generic findMax, with a function object, Version #1.
2 // Precondition: a.size( ) > 0.
3 template <typename Object, typename Comparator>
4 const Object & findMax( const vector<Object> & arr, Comparator cmp )
5 {
6     int maxIndex = 0;
7
8     for( int i = 1; i < arr.size( ); i++ )
9         if( cmp.isLessThan( arr[ maxIndex ], arr[ i ] ) )
10             maxIndex = i;
11
12     return arr[ maxIndex ];
13 }
14
15 class CaseInsensitiveCompare
16 {
17     public:
18         bool isLessThan( const string & lhs, const string & rhs ) const
19             { return strcmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
20 };
21
22 int main( )
23 {
24     vector<string> arr( 3 );
25     arr[ 0 ] = "ZEBRA"; arr[ 1 ] = "alligator"; arr[ 2 ] = "crocodile";
26     cout << findMax( arr, CaseInsensitiveCompare( ) ) << endl;
27
28     return 0;
29 }

```

图1-22 使用函数对象作为findMax的第二个参数的最简单思想，输出为ZEBRA

就可以改成更容易理解的isLessThan，调用为isLessThan(x,y)。最后，我们可以提供一个没有函数对象的findMax的版本。其实现是使用标准库函数对象模板less（在头文件functional中定义）来生成一个函数对象，该对象强制执行默认的顺序。图1-23显示了使用更典型的、也有一点晦涩的C++例程的实现。

34 第4章中将给出一个需要对其存储的数据进行排序的类。我们将编写大量的使用Comparable的代码，并显示使用函数对象所必须做的调整。在本书的其他部分，我们尽量避免涉及函数对象的细节，这样可以使代码尽可能简单，而以后要添加函数对象也并不困难。

### 1.6.5 类模板的分离编译

和其他的规则类一样，类模板可以在声明的时候整体实现，也可以将接口从实现中分离出来。但是，曾经支持分离编译模板的编译器已经越来越弱化并且需要特殊的平台来支持。这样一来，在许多情况下，整个的类模板连同其实现都放置在一个单独的头文件中。常见的标准库的实现在实现类模板时遵循这个原则。

35 附录描述了模板分离编译的机制。类模板接口的声明就是所期望的结果：以分号结束的成员函数，而不是一个实现。但是，正如附录所示，成员函数的实现会导致看起来很复杂的语法，特别是像operator=一样复杂的函数。更坏的情况是，在进行编译的时候，编译器经常报出函数丢失的信息。要避免这个问题需要特殊的平台来解决。

因此，在本书的联机代码中，所有类模板的实现都是在声明中整体实现而且是在单一的头文件中。这么做是因为看起来这是避免贯穿平台的编译问题的唯一方法。在本书中，为简单起见在

示例代码中提供类的接口时，假定分离编译工作良好，但是实现还是和联机代码中的是一样的。在特殊的平台方式下，如需要，可以机械地将我们的单一头文件实现转化成分离编译实现。可能会用到的一些情形可以参见附录。

```

1 // Generic findMax, with a function object, C++ style.
2 // Precondition: a.size( ) > 0.
3 template <typename Object, typename Comparator>
4 const Object & findMax( const vector<Object> & arr, Comparator isLessThan )
5 {
6     int maxIndex = 0;
7
8     for( int i = 1; i < arr.size( ); i++ )
9         if( isLessThan( arr[ maxIndex ], arr[ i ] ) )
10             maxIndex = i;
11
12     return arr[ maxIndex ];
13 }
14
15 // Generic findMax, using default ordering.
16 #include <functional>
17 template <typename Object>
18 const Object & findMax( const vector<Object> & arr )
19 {
20     return findMax( arr, less<Object>( ) );
21 }
22
23 class CaseInsensitiveCompare
24 {
25 public:
26     bool operator( )( const string & lhs, const string & rhs ) const
27     { return strcmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
28 };
29
30 int main( )
31 {
32     vector<string> arr( 3 );
33     arr[ 0 ] = "ZEBRA"; arr[ 1 ] = "alligator"; arr[ 2 ] = "crocodile";
34     cout << findMax( arr, CaseInsensitiveCompare( ) ) << endl;
35     cout << findMax( arr ) << endl;
36
37     return 0;
38 }

```

图1-23 使用C++风格的第二个版本的findMax的函数对象，输出为ZEBRA，然后是crocodile

## 1.7 使用矩阵

第10章的几个算法使用了通常称为矩阵的二维数组。C++类没有提供matrix类。但是一个合理的matrix类可以很快就写出来。基本的思想是使用向量的向量。这样做需要附加的操作符重载的知识。对matrix定义operator[]，即数组索引操作符。图1-24例举了matrix类。

### 1.7.1 数据成员、构造函数和基本访问函数

矩阵通过被声明为vector<Object>的vector的array数据成员来表述。构造函数首先构造array，通过rows来进入每个通过零参数构造函数构造的vector<Object>。这样我们就得到了



Object的零长度向量rows。

然后，进入构造函数代码体中，每一行的大小都被重新定义为cols列。这样构造函数就完成了，而此时的构造函数看起来就是一个二维数组。于是numrows和numcols访问函数就可以简单地实现了。

```

1  #ifndef MATRIX_H
2  #define MATRIX_H
3
4  #include <vector>
5  using namespace std;
6
7  template <typename Object>
8  class matrix
9  {
10     public:
11         matrix( int rows, int cols ) : array( rows )
12         {
13             for( int i = 0; i < rows; i++ )
14                 array[ i ].resize( cols );
15         }
16
17         const vector<Object> & operator[]( int row ) const
18         { return array[ row ]; }
19         vector<Object> & operator[]( int row )
20         { return array[ row ]; }
21
22         int numrows( ) const
23         { return array.size( ); }
24         int numcols( ) const
25         { return numrows( ) ? array[ 0 ].size( ) : 0; }
26     private:
27         vector< vector<Object> > array;
28 };
29
30 #endif

```

图1-24 一个完整的matrix类

### 1.7.2 operator[]

operator[]的思想如下：如果有一个matrix m，那么m[i]就应该返回一个对应matrix m的行i的向量。这样一来，m[i][j]就可以通过正常的vector索引操作给出向量m[i]的位置j。因此，matrix operator[]不是返回一个Object，而是一个vector<Object>。

operator[]应该返回Vector<Object>的实体。返回应该用值、引址还是常量引用呢？值返回立刻就可以排除掉，因为返回的实体太大了，尽管值返回可以保证调用结束后返回值肯定存在。接下来考虑引址返回和常量引用返回。考虑下面的方法（忽略混淆和大小不兼容的可能性，这两者都不影响算法）。

```

void copy( const matrix<int> & from, matrix<int> & to)
{
    for( int i=0; i<to.numrows(); i++)
        to[i] = from[i];
}

```

copy函数试图复制matrix from中的每一行到matrix to中的相应的行。很明显，如果operator[]返回常量引用，那么to[i]就不能出现在复制语句的左边。这样，看起来operator[]

应该是返回引用。然而，如果真这样做的话，在编译中就会有诸如`from[i]=to[i]`的表达式出现。因为虽然`from`是常量矩阵，但是`from[i]`不是常量向量。这在优秀的设计中是不允许的。

因此实际需要的是`operator[]`返回一个常量引用给`from`，但是返回一个普通引用给`to`。换句话说，我们需要两个版本的`operator[]`，它们的区别仅仅在于返回类型的不同。这是不允许的。然而，这里有一个漏洞：既然成员函数的定常性（即它到底是访问函数还是修改函数）是签名的一部分，我们可以使访问函数版本的`operator[]`返回常量引用，而修改函数版本的则返回简单引用。如图1-24所示，这样一来所有问题就都解决了。

### 1.7.3 析构函数、复制赋值和复制构造函数

因为`vector`处理了上述函数，所以，这些函数都是自动处理的。因此，这些函数就是完全功能的`matrix`类所需要的所有代码。

## 小结

这一章是本书后续部分的基础。面临大量输入时，一个算法所需要的运行时间是判断其好坏的重要标准。（当然，正确性是最重要的。）速度是相对的。对于某个问题在某台机器上运行很快的算法，对另一个问题或在另一台机器上运行就可能很慢。下一章中我们将讨论这些问题，并使用在这里讨论的数学知识来建立正式的模型。

## 练习

- 1.1 编写一个程序解决选择问题。令 $k = N/2$ 。画出表格显示你的程序对 $N$ 取不同值时的运行时间。
- 1.2 编写一个程序求解字谜游戏问题。
- 1.3 只使用处理I/O的`printDigit`，编写一个函数来输出任意`double`型量（可以是负的）。
- 1.4 C++提供形如`#include filename`的语句，它将`filename`读入并将其插入到`include`语句处。`include`语句可以嵌套；换句话说，文件`filename`本身还可以包含`include`语句，但是显然一个文件在任何链接中都不能包含它自己。编写一个程序，来读入一个文件，并输出该文件被`include`语句修改后的内容。
- 1.5 编写一个递归方法，它返回数 $N$ 的二进制表示中1的个数。利用这样的事实：如果 $N$ 是奇数，那么它等于 $N/2$ 的二进制表示中1的个数加1。
- 1.6 编写带有下列声明的例程：
 

```
void permute( const string & str);
void permute( const string & str, int low, int high);
```
- 1.7 证明下列公式：
  - a.  $\log X < X$  对所有的 $X > 0$ 成立。
  - b.  $\log(A^B) = B \log A$
- 1.8 计算下列各和：

a.  $\sum_{i=0}^{\infty} \frac{1}{4^i}$

b.  $\sum_{i=0}^{\infty} \frac{i}{4^i}$

$$*c. \sum_{i=0}^{\infty} \frac{i^2}{4^i}$$

$$**d. \sum_{i=0}^{\infty} \frac{i^N}{4^i}$$

1.9 估计

$$\sum_{i=\lfloor N/2 \rfloor}^N \frac{1}{i}$$

的值。

\*1.10  $2^{100} \pmod{5}$  是多少?

1.11 令  $F_i$  是在1.2节中定义的斐波那契数。证明下列各式:

$$a. \sum_{i=1}^{N-2} F_i = F_N - 2$$

$$b. F_N < \phi^N, \text{ 其中 } \phi = (1 + \sqrt{5})/2$$

\*\*c. 给出  $F_N$  准确的封闭形式的表达式。

1.12 证明下列公式:

$$a. \sum_{i=1}^N (2i-1) = N^2$$

$$b. \sum_{i=1}^N i^3 = \left( \sum_{i=1}^N i \right)^2$$

1.13 设计一个类模板 `Collection`, 来存储 `Object` 对象的集合 (在一个数组中), 以及该集合当前的大小。提供 `public` 型的函数 `isEmpty`、`makeEmpty`、`insert`、`remove` 和 `contains`。当且仅当该集合中存在等于 `x` 的一个 `Object` 时, `contains(x)` 返回 `true`。

1.14 设计一个类模板 `OrderedCollection`, 来存储 `Comparable` 的对象的集合 (在一个数组中), 以及该集合当前的大小。提供 `public` 型的函数 `isEmpty`、`makeEmpty`、`insert`、`remove`、`findMin` 和 `findMax`。`findMin` 和 `findMax` 分别返回该集合中最小的和最大的 `Comparable` 对象。说明: 如果这些操作发生在空集合里应该怎样处理。

1.15 使用图1-23的 `findMax` 例程, 编写一个 `main` 函数来找到按字母顺序排列时名字在最后面的 `Employee`。忽略大小写。

40

1.16 给 `matrix` 类添加一个 `resize` 成员函数和一个零参数构造函数。

## 参考文献

有许多好的教科书涵盖了本章所复习的数学内容, 其中的一小部分为[1]、[2]、[3]、[9]、[14]和[16]。参考文献[9]是特别配合算法分析的教材, 它是本书时常引用的三卷丛书系列的第一卷。更深入的材料在[6]中。

本书中将假设读者具备C++的知识。更多的内容参见[15]。该书由C++的创始人所编写, 描述了C++的最新设计标准, 具有最高的权威性。另一本标准的参考文献是[10]。C++的高级编程在[5]中讨论。分成两本的套书[11,12]详尽地讨论了C++的许多缺陷。本书中广泛研究的标准模板库可以参阅[13]。1.4节~1.7节中的材料可以作为我们将在本书中用到的一些要点的概括。我们还假设读者熟悉指针和递归 (本章中关于递归的总结是对递归的快速回顾), 在书中适当的地方我们将提供使用它们的一些提示。不熟悉递归的读者应该参考[17]或任何一本好的中等水平的程序设计教材。

一般的程序设计风格在多本书里均有所讨论, 其中一些经典的文献包括[4]、[7]和[8]。

1. M. O. Albertson and J. P. Hutchinson, *Discrete Mathematics with Algorithms*, John Wiley & Sons, New York, 1988.



2. Z. Bavel, *Math Companion for Computer Science*, Reston Publishing Co., Reston, Va., 1982.
3. R. A. Brualdi, *Introductory Combinatorics*, North-Holland, New York, 1977.
4. E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, N.J., 1976.
5. B. Eckel, *Thinking in C++*, Prentice Hall, Englewood Cliffs, N.J., 2nd ed. 2002.
6. R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, Mass., 1989.
7. D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.
8. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*, 2nd ed., McGraw-Hill, New York, 1978.
9. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed., Addison-Wesley, Reading, Mass., 1997.
10. S. B. Lippman and J. Lajoie, *C++ Primer*, 3rd ed., Addison-Wesley, Reading, Mass., 1998.
11. S. Meyers, *50 Specific Ways to Improve Your Programs and Designs*, 2nd ed., Addison-Wesley, Reading, Mass., 1998.
12. S. Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, Reading, Mass., 1996.
13. D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison-Wesley, Reading, Mass., 1996.
14. F. S. Roberts, *Applied Combinatorics*, Prentice Hall, Englewood Cliffs, N.J., 1984.
15. B. Stroustrup, *The C++ Programming Language*, 3rd ed., Addison-Wesley, Reading, Mass., 1997.
16. A. Tucker, *Applied Combinatorics*, 2nd ed., John Wiley & Sons, New York, 1984.
17. M. A. Weiss, *Algorithms, Data Structures, and Problem Solving with C++*, 2nd ed., Addison-Wesley, Reading, Mass., 2000.

**算**法 (algorithm) 是为求解一个问题需要遵循的、被清楚地指定的简单指令的集合。对于一个问题，一旦某种算法给定并且 (以某种方式) 被确定是正确的，那么重要的一步就是确定该算法将需要多少诸如时间或空间等资源量的问题。如果一个问题的求解算法竟然需要长达一年时间，那么这种算法就很难能有什么用处。同样，一个需要几吉字节 (gigabyte) 内存的算法在当前的大多数机器上也是没法使用的。

本章将讨论

- 如何估计一个程序所需要的时间。
- 如何将一个程序的运行时间从天或年降低到不足一秒。
- 粗心地使用递归的后果。
- 用于将一个数自乘得到其幂以及计算两个数的最大公因数的非常有效的算法。

## 2.1 数学基础

估计算法资源消耗所需的分析一般说来是一个理论问题，因此需要一套正式的系统构架。我们先从某些数学定义开始。

本书将使用下列四个定义。

**定义2.1** 如果存在正常数  $c$  和  $n_0$  使得当  $N \geq n_0$  时  $T(N) \leq cf(N)$ ，则记为  $T(N) = O(f(N))$ 。

**定义2.2** 如果存在正常数  $c$  和  $n_0$  使得当  $N \geq n_0$  时  $T(N) \geq cg(N)$ ，则记为  $T(N) = \Omega(g(N))$ 。

**定义2.3**  $T(N) = \Theta(h(N))$  当且仅当  $T(N) = O(h(N))$  和  $T(N) = \Omega(h(N))$ 。

**定义2.4** 如果对所有的常数  $c$  存在  $n_0$  使得当  $N > n_0$  时  $T(N) < cp(N)$ ，则记为  $T(N) = o(p(N))$ 。非正式的定义为：如果  $T(N) = O(p(N))$  且  $T(N) \neq \Theta(p(N))$ ，则  $T(N) = o(p(N))$ 。

这些定义的目的是要在函数间建立一种相对的级别。给定两个函数，通常存在一些点，在这些点上一个函数的值小于另一个函数的值，因此，像  $f(N) < g(N)$  这样的声明是没有什么意义的。于是，我们比较它们的**相对增长率** (relative rates of growth)。当将相对增长率应用到算法分析的时候，我们将会明白为什么它是重要的度量。

虽然  $N$  较小时， $1000N$  要比  $N^2$  大，但  $N^2$  以更快的速度增长，因此  $N^2$  最终将更大。在这种情况下， $N=1000$  是转折点。第一个定义是说，最后总会存在某个点  $n_0$ ，从它以后  $c \cdot f(N)$  总是至少与  $T(N)$  一样大，从而若忽略常数因子，则  $f(N)$  至少与  $T(N)$  一样大。在我们的例子中， $T(N) = 1000N$ ， $f(N) = N^2$ ， $n_0 = 1000$  而  $c = 1$ 。我们也可以让  $n_0 = 10$  而  $c = 100$ 。因此，我们可以说  $1000N = O(N^2)$  ( $N$  平方级)。这种记法称为**大O记法**。人们常常不说“……级的”，而是说“大O……”。

如果我们用传统的不等式来比较增长率，那么第一个定义是说 $T(N)$ 的增长率小于等于( $\leq$ ) $f(N)$ 的增长率。第二个定义 $T(N) = \Omega(g(N))$ （念成“omega”）是说 $T(N)$ 的增长率大于等于( $\geq$ ) $g(N)$ 的增长率。第三个定义 $T(N) = \Theta(h(N))$ （念成“theta”）是说 $T(N)$ 的增长率等于( $=$ ) $h(N)$ 的增长率。最后一个定义 $T(N) = o(p(N))$ （念成“小o”）说的则是 $T(N)$ 的增长率小于( $<$ ) $p(N)$ 的增长率。它不同于大O，因为大O包含增长率相同这种可能性。

为了证明某个函数 $T(N) = O(f(N))$ ，我们通常不是形式地使用这些定义，而是使用一些已知的结果。一般说来，这就意味着证明（或确定假设不成立）是非常简单的计算而不应涉及微积分，除非是遇到特殊的情况（这几乎不可能发生在算法分析中）。

当我们说 $T(N) = O(f(N))$ 时，我们是在保证函数 $T(N)$ 是在以不快于 $f(N)$ 的速度增长；因此 $f(N)$ 是 $T(N)$ 的一个上界（upper bound）。由于这意味着 $f(N) = \Omega(T(N))$ ，于是我们说 $T(N)$ 是 $f(N)$ 的一个下界（lower bound）。

作为一个例子， $N^3$ 增长比 $N^2$ 快，因此我们可以说 $N^2 = O(N^3)$ 或 $N^3 = \Omega(N^2)$ 。 $f(N) = N^2$ 和 $g(N) = 2N^2$ 以相同的速率增长，从而 $f(N) = O(g(N))$ 和 $f(N) = \Omega(g(N))$ 都是正确的。当两个函数以相同的速率增长时，是否需要使用记号 $\Theta()$ 表示取决于具体的上下文。直观地说，如果 $g(N) = 2N^2$ ，那么 $g(N) = O(N^4)$ ， $g(N) = O(N^3)$ 和 $g(N) = O(N^2)$ 从技术上看都是成立的，但最后一个选择是最好的答案。写法 $g(N) = \Theta(N^2)$ 不仅表示 $g(N) = O(N^2)$ ，而且还表示结果尽可能地好（严格）。

我们需要掌握的重要结论为：

法则1 如果 $T_1(N) = O(f(N))$ 且 $T_2(N) = O(g(N))$ ，那么

(a)  $T_1(N) + T_2(N) = O(f(N) + g(N))$ （直观地非正式地表达为 $\max(O(f(N)), O(g(N)))$ ）

(b)  $T_1(N) T_2(N) = O(f(N) g(N))$

法则2 如果 $T(N)$ 是一个 $k$ 次多项式，则  $T(N) = \Theta(N^k)$ 。

44

法则3 对任意常数 $k$ ， $\log^k N = O(N)$ 。它告诉我们对数增长得非常缓慢。

这些信息足以按照增长率对大部分常见的函数进行分类（见图2-1）。

函数	名称
$c$	常量
$\log N$	对数
$\log^2 N$	对数的平方
$N$	线性
$N \log N$	
$N^2$	二次
$N^3$	三次
$2^N$	指数

图2-1 典型的增长率

需要注意以下几点。首先，将常数或低阶项放进大O是非常不好的习惯。不要说 $T(N) = O(2N^2)$ 或 $T(N) = O(N^2 + N)$ 。在这两种情形下，正确的形式是 $T(N) = O(N^2)$ 。这就是说，在需要大O表示的任何分析中，各种简化都是可能发生的。低阶项一般可以被忽略，而常数也可以丢弃掉。此时，要求的精度是很低的。

其次，我们总能够通过计算极限 $\lim_{N \rightarrow \infty} f(N)/g(N)$ 来确定两个函数 $f(N)$ 和 $g(N)$ 的相对增长率，



必要的时候可以使用洛必达法则<sup>1</sup>。该极限可以有四种可能的值：

- 极限是0：这意味着 $f(N) = o(g(N))$ 。
- 极限是 $c \neq 0$ ：这意味着 $f(N) = \Theta(g(N))$ 。
- 极限是 $\infty$ ：这意味着 $g(N) = o(f(N))$ 。
- 极限摆动：二者无关（本书中不会发生这种情形）。

使用这种方法几乎总能够算出相对增长率，不过有些复杂化。通常，两个函数 $f(N)$ 和 $g(N)$ 间的关系可以用简单的代数方法得到。例如，如果 $f(N) = N \log N$ 和 $g(N) = N^{1.5}$ ，那么为了确定 $f(N)$ 和 $g(N)$ 哪个增长得更快，实际上就是确定 $\log N$ 和 $N^{0.5}$ 哪个增长更快。这与确定 $\log^2 N$ 和 $N$ 哪个增长更快是一样的，而后者是个简单的问题，因为我们已经知道， $N$ 的增长要快于 $\log$ 的任意的幂。因此， $g(N)$ 的增长快于 $f(N)$ 的增长。

45

另外，在风格上还应注意：不要说成 $f(N) \leq O(g(N))$ ，因为定义已经隐含有不等式了。写成 $f(N) \geq O(g(N))$ 是错误的，它没有意义。

作为典型的分析的例子，考虑在互连网上下载一个文件的问题。设有初始3 s的延迟（来建立连接），此后下载以1.5 KB/s进行。可以推出，如果文件为 $N$  KB，下载时间由公式 $T(N) = N/1.5 + 3$ 表示。这是一个线性函数（linear function）。注意，下载一个1500 KB的文件所用时间（1003 s）近似（但不是精确地）为下载750 KB文件所用时间（503 s）的2倍。这是典型的线性函数。还要注意，如果连接的速度快2倍，那么两个文件的下载时间都要减少，但1500 KB的文件的下载时间仍然近似为750 KB文件的下载时间的2倍。这是线性时间算法的典型特点，这就是我们写 $T(N) = O(N)$ 而忽略常数因子的原因。（虽然使用大 $\Theta$ 会更精确，但是一般给出的是大 $O$ 答案。）还要看到，这种做法不是对所有的算法都成立。对于1.1节描述的第一个选择算法，运行时间由执行一次排序所花费的时间来决定。对诸如所提出的冒泡排序这样的简单排序算法，当输入量增加到2倍的时候，对大量输入来说，运行时间则增加到4倍。这是因为这些算法不是线性的。在讨论排序时，我们将会看到，不好的排序算法是 $O(N^2)$ ，或称为二次的。

## 2.2 模型

为了在形式的框架中分析算法，我们需要一个计算模型。我们的模型基本上是一台标准的计算机，在机器中指令被顺序地执行。该模型有一个标准的简单指令系统，如加法、乘法、比较和赋值等。但不同于实际计算机的是，模型机做任意一件简单的工作都恰好花费一个时间单位。为了合理起见，我们将假设我们的模型如同一台现代计算机那样有定长（比如32位）的整数并且不存在诸如矩阵求逆或排序运算，它们显然不能在一个时间单位内完成。我们还假设模型机有无限的内存。

显然，这个模型有些缺点。很明显，在现实生活中不是所有的运算都恰好花费相同的时间。特别是，在我们的模型中，一次磁盘读入按一次加法计时，即使加法一般要快几个数量级。还有，由于假设有无限的内存，所以不用担心缺页，但这可能是个实际问题，特别是对高效的算法。

## 2.3 要分析的问题

要分析的最重要的资源一般说来就是运行时间。有几个因素影响程序的运行时间。有些因

1. 洛必达法则(L'Hôpital's rule)说的是，若 $\lim_{N \rightarrow \infty} f(N) = \infty$ 且 $\lim_{N \rightarrow \infty} g(N) = \infty$ ，则 $\lim_{N \rightarrow \infty} f(N)/g(N) = \lim_{N \rightarrow \infty} f'(N)/g'(N)$ ，而 $f'(N)$ 和 $g'(N)$ 分别是 $f(N)$ 和 $g(N)$ 的导数。

素（如所使用的编译器和计算机）显然超出了任何理论模型的范畴，因此，虽然它们是重要的，但是我们在这里还是不能处理它们。剩下的主要因素则是所使用的算法以及对该算法的输入。

典型的情形是，输入的大小是主要的考虑方面。我们定义两个函数 $T_{\text{avg}}(N)$ 和 $T_{\text{worst}}(N)$ ，分别为算法对于输入 $N$ 所花费的平均情形的和最坏情形的运行时间。显然， $T_{\text{avg}}(N) \leq T_{\text{worst}}(N)$ 。如果存在多于一个的输入，那么这些函数可以有多个的变量。

偶尔也分析一个算法最好情形的性能。不过，通常这并不重要，因为它不代表典型的结果。平均情形性能常常反应典型的结果，而最坏情形的性能则代表对任何可能的输入在性能上的一种保证。还要注意，虽然在这一章我们分析C++程序，但所得到的界实际上是算法的界，而不是程序的界。程序是算法以一种特殊编程语言的实现，程序设计语言的细节几乎总是不影响大O的答案。如果一个程序比算法分析提出的速度慢，那么可能存在低效的实现。这种情况有可能发生在C++语言中，比如，数组被当作整体而漫不经心地进行拷贝，而不是由引用来传递。在12.7节的最后两段有一个极其巧妙的例子说明了这个问题。因此，在未来各章我们将分析算法而不是分析程序。

一般说来，若无特别说明，则所需要的量就是最坏情况的运行时间。其原因之一是它对所有的输入提供了一个界限，包括特别坏的输入，而平均情况分析不提供这样的界。另一个原因是平均情况的界计算起来通常要困难得多。在某些情况下，“平均”的定义可能影响分析的结果。（例如，什么是下述问题的平均输入？）

作为一个例子，我们将在下一节考虑下述问题：

**最大的子序列和问题** 给定整数 $A_1, A_2, \dots, A_N$ （可能有负数），求 $\sum_{k=i}^j A_k$ 的最大值。（为方便起见，如果所有整数均为负数，则最大子序列和为0。）

例如：对于输入-2, 11, -4, 13, -5, -2，答案为20（从 $A_2$ 到 $A_4$ ）。

这个问题之所以有吸引力，主要是因为存在求解它的很多算法，而这些算法的性能又差异很大。我们将讨论求解该问题的四种算法。这四种算法在某台计算机上（究竟是哪一台具体的计算机是不重要的）的运行时间在图2-2给出。

输入 大小	算法时间			
	1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 10$	0.000 009	0.000 004	0.000 006	0.000 003
$N = 100$	0.002 580	0.000 109	0.000 045	0.000 006
$N = 1\,000$	2.281 013	0.010 203	0.000 485	0.000 031
$N = 10\,000$	NA	1.232 9	0.005 712	0.000 317
$N = 100\,000$	NA	135	0.064 618	0.003 206

图2-2 对于计算最大子序列和的几种算法的运行时间（s）

在图中有几个重要的情况值得注意。对于小量的输入，算法瞬间就完成了。因此如果只是小量输入的情形，那么花费大量的努力去设计聪明的算法恐怕就太不值得了。另一方面，近来对于重写那些不再合理的基于小输入量假设而在五年以前编写的程序确实存在着巨大的市场。现在看来，这些程序太慢了，因为它们用的是些不好的算法。对于大量的输入，算法4显然是最好的选择（虽然算法3也是可用的）。

其次，图中所给出的时间不包括读入数据所需要的时间。对于算法4，仅仅从磁盘读入数据

47 所用的时间很可能在数量级上比求解上述问题所需要的时间还要大。这是许多高效算法中的典型特点。数据的读入一般是个瓶颈；一旦数据读入，问题就会迅速解决。但是，对于低效率的算法，情况就不同了，它必然要耗费大量的计算机资源。因此只要可能，使得算法足够高效而不至于成为问题的瓶颈是非常重要的。

48 图2-3指出这四种算法运行时间的增长率。尽管该图只包含 $N$ 从10到100的值，但是相对增长率还是很明显的。虽然算法3的图看起来是线性的，但是用直尺的边（或是一张纸）就容易验证它并不是直线。图2-4显示对于更大值的性能。该图戏剧性地描述出，即使是适度大小的输入量，低效算法依然无用。

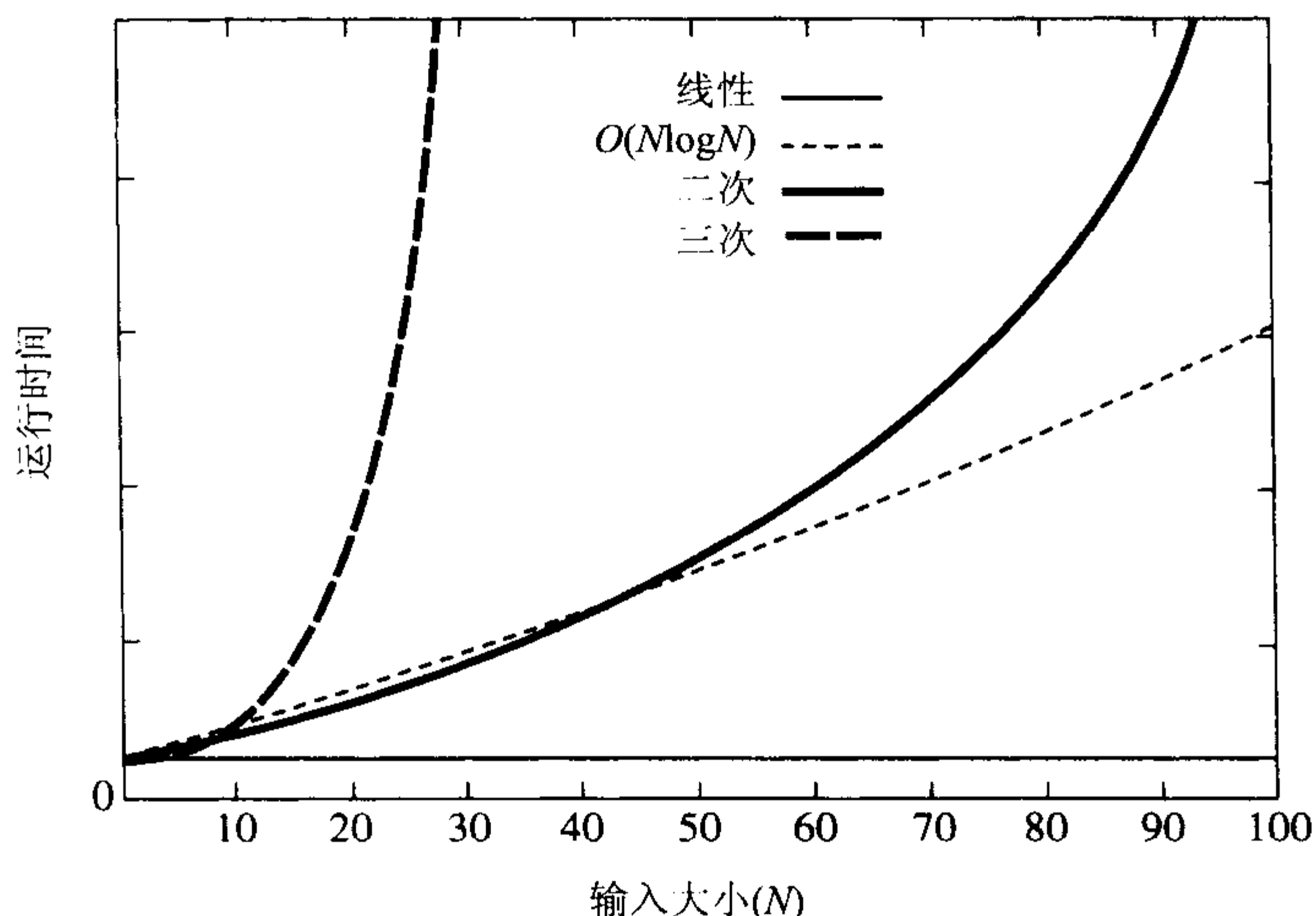


图2-3 各种计算最大子序列和的算法图（横坐标为 $N$ ，纵坐标为时间）

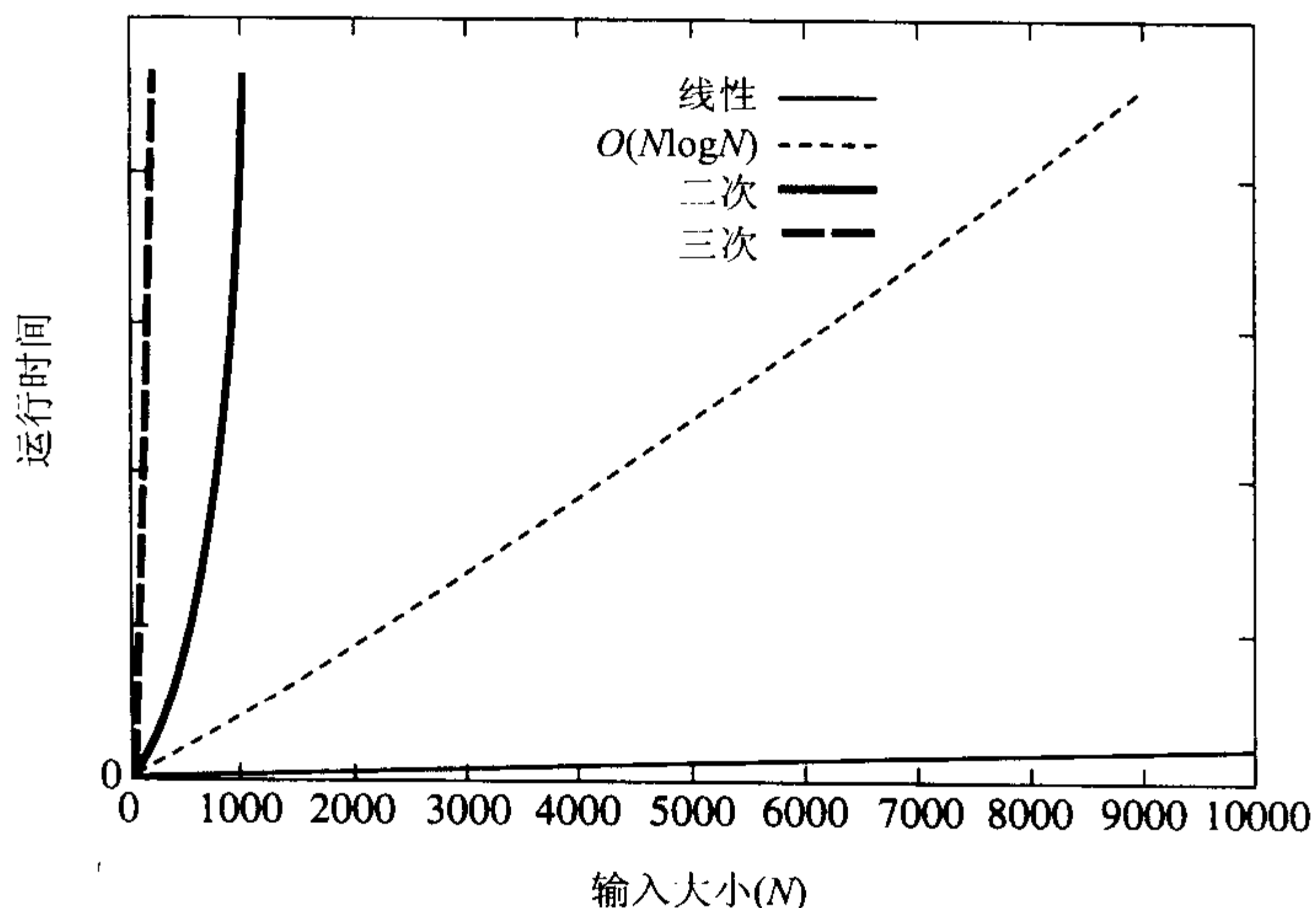


图2-4 各种计算最大子序列和的算法图（横坐标为 $N$ ，纵坐标为时间）

## 2.4 运行时间计算

有几种方法可以用来估计一个程序的运行时间。前面的表是凭经验得到的。如果认为两个程序花费大致相同的时间，要确定哪个程序更快的最好方法很可能就是将它们编码并运行！



一般地，存在几种算法思想，而我们总愿意尽早去除那些不好的算法思想，因此，通常需要对算法进行分析。不仅如此，进行分析的能力还有助于洞察到如何设计高效算法。一般说来，分析还能准确地确定需要仔细编码的瓶颈。

为了简化分析，我们将采纳如下的约定：不存在特定的时间单位。因此，我们抛弃那些常数系数。我们还将抛弃低阶项，因此所要做的是计算大O运行时间。由于大O是一个上界，因此我们必须仔细，绝不要低估程序的运行时间。实际上，分析的结果是程序在一定的时间范围内终止运行的保证。程序可能提前结束，但绝不可能拖后。

### 2.4.1 一个简单的例子

这里是计算  $\sum_{i=1}^N i^3$  的一个简单的程序片段：

```
int sum ( int n )
{
    int partialSum;

1   partialSum = 0;
2   for( int i = 1; i <= n; i++)
3       partialSum += i * i * i;
4   return partialSum;
}
```

这个程序的分析是简单的。声明不计时间。第一行和第四行各占1个时间单位。第三行每执行一次占用4个时间单位（两个乘法、一次加法和一次赋值），而执行 $N$ 次共占用 $4N$ 个时间单位。第二行在初始化 $i$ 、测试 $i \leq N$ 和对 $i$ 的自增运算中隐含着开销。所有这些的总开销是初始化为1个时间单位，所有的测试为 $N+1$ 个时间单位，而所有的自增运算 $N$ 个时间单位，共 $2N+2$ 。我们忽略调用方法和返回值的开销，得到总量是 $6N+4$ 。因此，我们说该方法是 $O(N)$ 。

如果每次分析一个程序都要演示所有这些工作，那么这项任务很快就会变成不可行的工作。幸运的是，由于我们给出的是大O的结果，因此就存在许多可以采取的捷径并且不影响最后的结果。例如，第三行（每次执行时）显然是 $O(1)$ 语句，因此精确计算它究竟是2个、3个还是4个时间单位是愚蠢的；这无关紧要。第一行与for循环相比显然是不重要的，所以在这里花费时间也是不明智的。这使我们得到若干一般法则。

49

### 2.4.2 一般法则

**法则1: for循环** 一个for循环的运行时间至多是该for循环内语句（包括测试）的运行时间乘以迭代的次数。

**法则2: 嵌套循环** 从里向外分析这些循环。在一组嵌套循环内部的一条语句总的运行时间为该语句的运行时间乘以该组所有循环的大小的乘积。

作为一个例子，下列程序片段为 $O(N^2)$ ：

```
for( i = 0; i < n; i++)
    for( j = 0; j < n; j++)
        k++;
```

**法则3: 顺序语句** 将各个语句的运行时间求和即可（这意味着，其中的最大值就是所得的运行时间；见2.1节的法则1）。

作为一个例子，下面的程序片段先是花费 $O(N)$ ，接着是 $O(N^2)$ ，因此总量也是 $O(N^2)$ ：

```

for( i = 0; i < n; i++)
    a[i] = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < n; j++)
        a[i] += a[j] + i + j;

```

**50** 法则4: **If/Else**语句 对于程序片段

```

if ( condition )
    S1
else
    S2

```

一个if/else语句的运行时间从不超过判断再加上 $S_1$ 和 $S_2$ 中运行时间较长者的总的运行时间。

显然在某些情形下这么估计有些过高，但决不会估计过低。

其他的法则都是显然的，但是，分析的基本策略是从内部（或最深层部分）向外展开工作。如果有方法调用，那么这些调用要首先分析。如果有递归过程，那么存在几种选择。若递归实际上只是被薄面纱遮住的for循环，则分析通常是很简单的。例如，下面的方法实际上就是一个简单的循环，其运行时间为 $O(N)$ ：

```

long factorial ( int n )
{
    if ( n <= 1 )
        return 1;
    else
        return n * factorial ( n - 1 );
}

```

这个例子实际上对递归的使用并不恰当。当正常使用递归时，将其转换成一个简单的循环结构是相当困难的。在这种情况下，分析将涉及求解一个递推关系。为了观察到这种可能发生的情形，考虑下列程序，实际上它对递归的使用效率低得令人难以置信。

```

long fib ( int n )
{
1   if ( n <= 1 )
2       return 1;
    else
3       return fib ( n - 1 ) + fib ( n - 2 );
}

```

初看起来，该程序似乎对递归的使用非常聪明。可是，如果将程序编码，赋予 $N$ 大约40左右的值并运行，那么这个程序的效率低得吓人。分析是十分简单的。令 $T(N)$ 为函数fib( $n$ )的运行时间。如果 $N=0$ 或 $N=1$ ，则运行时间是某个常数值，即第一行上做判断以及返回所用的时间。因为常数并不重要，所以我们可以说 $T(0)=T(1)=1$ 。对于 $N$ 的其他值的运行时间则相对于基准情形的运行时间来度量。若 $N>2$ ，则执行该函数的时间是第一行上的常数时间加上第三行上的时间。第三行由一次加法和两次函数调用组成。由于函数调用不是简单的运算，必须通过它们自身来分析。

**51** 第一次函数调用是fib( $n-1$ )，从而按照 $T$ 的定义它需要 $T(N-1)$ 个时间单位。类似的论证指出，第二次方法调用需要 $T(N-2)$ 个时间单位。此时总的时间需求为 $T(N-1)+T(N-2)+2$ ，其中2指的是第一行上的时间加上第三行上的加法的时间。于是对于 $N\geq 2$ 我们有下列关于fib( $n$ )的运行时间公式：

$$T(N) = T(N-1) + T(N-2) + 2$$

但是 $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ ，因此由归纳法容易证明 $T(N) \geq \text{fib}(n)$ 。在1.2.5节我们证明过 $\text{fib}(N) < (5/3)^N$ ，类似的计算可以证明（对于 $N > 4$ ） $\text{fib}(N) \geq (3/2)^N$ ，从而这个程序的运行时间以指数的速度增长。这大致是最坏的情况。通过使用一个简单的数组并使用一个for循环，运行时间可以被实质性地减下来。

这个程序之所以缓慢，是因为存在大量多余的工作要做，违反了在1.3节中叙述的递归的第4条主要的法则（合成效益法则）。注意，在第三行上的第一次调用即 $\text{fib}(N-1)$ 实际上同时计算了 $\text{fib}(N-2)$ 。这个信息被抛弃而在第三行上的第二次调用时又重新计算了一遍。抛弃的信息量递归地合成起来并导致巨大的运行时间。这或许是格言“计算任何事情不要超过一次”的最好实例，但它不应成为远离递归而不敢使用它的理由。本书中我们将随处可见递归的出色的使用。

### 2.4.3 最大子序列和问题的解

现在我们将要叙述四个算法来求解早先提出的最大子序列和的问题。第一个算法在图2-5中表述，它只是穷举式地尝试所有的可能。for循环中的循环变量反映C++中数组从0开始而不是从1开始这样一个事实。再有，本算法并不计算实际的子序列；实际的计算还要添加一些额外的代码。

```

1  /**
2   * Cubic maximum contiguous subsequence sum algorithm.
3   */
4  int maxSubSum1( const vector<int> & a )
5  {
6      int maxSum = 0;
7
8      for( int i = 0; i < a.size( ); i++ )
9          for( int j = i; j < a.size( ); j++ )
10             {
11                 int thisSum = 0;
12
13                 for( int k = i; k <= j; k++ )
14                     thisSum += a[ k ];
15
16                 if( thisSum > maxSum )
17                     maxSum = thisSum;
18             }
19
20     return maxSum;
21 }
```

图2-5 算法1

该算法肯定会正确运行（这不应该花太多的时间去证明）。运行时间为 $O(N^3)$ ，这完全取决于第13行和第14行，第14行由一个隐含于三重嵌套for循环中的 $O(1)$ 语句组成。第8行上的循环大小为 $N$ 。

第2个循环大小为 $N-i$ ，它可能要小，但也可能是 $N$ 。我们必须假设最坏的情况，而这可能会使得最终的界有些大。第3个循环的大小为 $j-i+1$ ，我们也要假设它的大小为 $N$ 。因此总数为 $O(1 \cdot N \cdot N \cdot N) = O(N^3)$ 。语句6总共的开销只是 $O(1)$ ，而语句16和17也只不过总共开销 $O(N^2)$ ，因为它们只是两层循环内部的简单表达式。

事实上，考虑到这些循环的实际大小，更精确的分析指出答案是 $\Theta(N^3)$ ，而我们上面的估计高出个因子6（不过这并无大碍，因为常数不影响数量级）。一般说来，在这类问题中上述结论是



正确的。精确的分析由和  $\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^j 1$  得到，该和指出程序的第14行被执行了多少次。使用1.2.3节中的公式可以对该和从内到外求值。特别地，我们将用到前 $N$ 个整数求和以及前 $N$ 个平方数求和的公式。首先我们有

52

$$\sum_{k=i}^j 1 = j - i + 1$$

接着，我们得到

$$\sum_{j=i}^{N-1} (j - i + 1) = \frac{(N - i + 1)(N - i)}{2}$$

这个和数是对前 $N - i$ 个整数求和而算得。为完成全部计算，我们有

$$\begin{aligned} \sum_{i=0}^{N-1} \frac{(N - i + 1)(N - i)}{2} &= \sum_{i=1}^N \frac{(N - i + 1)(N - i + 2)}{2} \\ &= \frac{1}{2} \sum_{i=1}^N i^2 - \left(N + \frac{3}{2}\right) \sum_{i=1}^N i + \frac{1}{2} (N^2 + 3N + 2) \sum_{i=1}^N 1 \\ &= \frac{1}{2} \frac{N(N+1)(2N+1)}{6} - \left(N + \frac{3}{2}\right) \frac{N(N+1)}{2} + \frac{N^2 + 3N + 2}{2} N \\ &= \frac{N^3 + 3N^2 + 2N}{6} \end{aligned}$$

53

我们可以通过撤除一个for循环来避免三次运行时间。不过这并不总是可能的，在这个例子中，算法中出现了大量不必要的计算。纠正这种低效率的改进算法可以通过观察  $\sum_{k=i}^j A_k = A_j + \sum_{k=i}^{j-1} A_k$  而看出，因此算法1中第13行和第14行上的计算过度地耗费了时间。图2-6是一种改进的算法。算法2显然是 $O(N^2)$ ；对它的分析甚至比前面的分析还要简单。

```

1  /**
2   * Quadratic maximum contiguous subsequence sum algorithm.
3   */
4  int maxSubSum2( const vector<int> & a )
5  {
6      int maxSum = 0;
7
8      for( int i = 0; i < a.size( ); i++ )
9      {
10         int thisSum = 0;
11         for( int j = i; j < a.size( ); j++ )
12         {
13             thisSum += a[ j ];
14
15             if( thisSum > maxSum )
16                 maxSum = thisSum;
17         }
18     }
19
20     return maxSum;
21 }
```

图2-6 算法2

对这个问题有一个递归和相对复杂的 $O(N \log N)$ 解法，现在对其进行论述。要是真的没出现 $O(N)$ （线性的）解法，这个算法就会是体现递归威力的极好的范例了。该方法采用一种“分治”

(divide-and-conquer) 策略。其想法是把问题分成两个大致相等的子问题，然后递归地对它们求解，这是“分”的部分。“治”阶段将两个子问题的解合并到一起并可能再做少量的附加工作，最后得到整个问题的解。

在我们的例子中，最大子序列和可能出现在三处地方：或者整个出现在输入数据的左半部，或者整个出现在右半部，或者跨越输入数据的中部从而占据左右两半部分。前两种情况可以递归求解。第三种情况的最大和可以通过求出前半部分的最大和（包含前半部分的最后一个元素）以及后半部分的最大和（包含后半部分的第一个元素）而得到。然后将这两个和加在一起。例如，考虑下列输入：

前半部分					后半部分		
4	-3	5	-2	-1	2	6	-2

54

其中前半部分的最大子序列和为6（从元素 $A_1$ 到 $A_3$ ）而后半部分的最大子序列和为8（从元素 $A_6$ 到 $A_7$ ）。

前半部分包含其最后一个元素的最大和是4（从元素 $A_1$ 到 $A_4$ ），而后半部分包含其第一个元素的最大和是7（从元素 $A_5$ 到 $A_7$ ）。因此，横跨这两部分且通过中间的最大和为 $4 + 7 = 11$ （从元素 $A_1$ 到 $A_7$ ）。

我们看到，在形成本例中的最大子序列和的三种方法中，最好的方法是包含两部分的元素。于是，答案为11。图2-7提出了这种策略的一种实现手段。

有必要对算法3的程序进行一些说明。递归函数调用的一般形式是传递输入数组以及左边界和右边界，它们界定了数组要被处理的部分。单行驱动程序通过传递数组以及边界0和 $N-1$ 而启动该函数。

第8行至第12行处理基准情况。如果 $left == right$ ，那么只有一个元素，并且当该元素非负时它就是最大子序列。 $left > right$ 的情况是不可能出现的，除非 $N$ 是负数（尽管程序中的小的扰动有可能致使这种混乱产生）。第15行和第16行执行两个递归调用。我们可以看到，递归调用总是用于小于原问题的问题，不过程序中的小扰动有可能破坏这个特性。第18行至第24行以及第26行至第32行计算达到中间分界处的两个最大的和数。这两个值的和为横跨左右两边的最大和。例程max3（未显示出）返回这三个可能的最大和中的最大者。

显然，算法3需要比前面两种算法更多的编程工作量。然而，程序短并不总是意味着程序好。正如我们在前面显示算法运行时间的表中已经看到的，除最小的输入外，该算法比前两个算法明显要快。

对运行时间的分析方法与在分析计算斐波那契数程序时的方法类似。令 $T(N)$ 是求解大小为 $N$ 的最大子序列和问题所花费的时间。如果 $N = 1$ ，则算法执行程序第8行到第12行花费某个常数时间量，我们称之为一个时间单位。于是， $T(1) = 1$ 。否则，程序必须运行两个递归调用，即在第19行和第32行之间的两个for循环，以及某个小的簿记量，如第14行和第34行。这两个for循环接触到子序列中的每一个元素，而在循环内部的工作量是常量，因此，在第19到第32行花费的时间为 $O(N)$ 。在第8行到第14行，第18、26和34行上的程序的工作量都是常量，从而与 $O(N)$ 相比可以忽略。其余就是第15行和第16行上运行的工作。这两行求解大小为 $N/2$ 的子序列问题（假设 $N$ 是偶数）。因此，这两行每行花费 $T(N/2)$ 个时间单位，共花费 $2T(N/2)$ 个时间单位。该算法花费的总的时间为 $2T(N/2) + O(N)$ 。我们得到方程组

$$\begin{aligned} T(1) &= 1 \\ T(N) &= 2T(N/2) + O(N) \end{aligned}$$

为了简化计算，我们可以用 $N$ 代替上面方程中的 $O(N)$ 项；由于 $T(N)$ 最终还是要用大 $O$ 来表示的，因此这么做并不影响答案。在第7章，我们将会看到如何严格地求解这个方程。这里进行粗略的推导，如果 $T(N) = 2T(N/2) + N$ ，且 $T(1) = 1$ ，那么 $T(2) = 4 = 2 \times 2$ ， $T(4) = 12 = 4 \times 3$ ， $T(8) = 32 = 8 \times 4$ ，以及 $T(16) = 80 = 16 \times 5$ 。显然的，可以从其形式上得到，若 $N = 2^k$ ，则 $T(N) = N \times (k + 1) = N \log N + N = O(N \log N)$ 。

```

1  /**
2   * Recursive maximum contiguous subsequence sum algorithm.
3   * Finds maximum sum in subarray spanning a[left..right].
4   * Does not attempt to maintain actual best sequence.
5   */
6  int maxSumRec( const vector<int> & a, int left, int right )
7  {
8      if( left == right ) // Base case
9          if( a[ left ] > 0 )
10             return a[ left ];
11         else
12             return 0;
13
14     int center = ( left + right ) / 2;
15     int maxLeftSum = maxSumRec( a, left, center );
16     int maxRightSum = maxSumRec( a, center + 1, right );
17
18     int maxLeftBorderSum = 0, leftBorderSum = 0;
19     for( int i = center; i >= left; i-- )
20     {
21         leftBorderSum += a[ i ];
22         if( leftBorderSum > maxLeftBorderSum )
23             maxLeftBorderSum = leftBorderSum;
24     }
25
26     int maxRightBorderSum = 0, rightBorderSum = 0;
27     for( int j = center + 1; j <= right; j++ )
28     {
29         rightBorderSum += a[ j ];
30         if( rightBorderSum > maxRightBorderSum )
31             maxRightBorderSum = rightBorderSum;
32     }
33
34     return max3( maxLeftSum, maxRightSum,
35                 maxLeftBorderSum + maxRightBorderSum );
36 }
37
38 /**
39 * Driver for divide-and-conquer maximum contiguous
40 * subsequence sum algorithm.
41 */
42 int maxSubSum3( const vector<int> & a )
43 {
44     return maxSumRec( a, 0, a.size( ) - 1 );
45 }

```

图2-7 算法3

这个分析假设 $N$ 是偶数，否则 $N/2$ 就不确定了。通过该分析的递归性质可知，实际上只有当 $N$ 是2的幂时结果才是合理的，否则我们最终要遇到大小不是偶数的子问题，方程就是无效的了。当 $N$ 不是2的幂的时候，我们需要更加复杂一些的分析，但是大 $O$ 的结果是不变的。



在后面的章节中，我们将看到递归的几个出色的应用。这里，继续介绍求解最大子序列和的第4种方法，该算法实现起来要比递归算法简单而且更为有效。它在图2-8中给出。

```

1  /**
2   * Linear-time maximum contiguous subsequence sum algorithm.
3   */
4  int maxSubSum4( const vector<int> & a )
5  {
6      int maxSum = 0, thisSum = 0;
7
8      for( int j = 0; j < a.size( ); j++ )
9      {
10         thisSum += a[ j ];
11
12         if( thisSum > maxSum )
13             maxSum = thisSum;
14         else if( thisSum < 0 )
15             thisSum = 0;
16     }
17
18     return maxSum;
19 }

```

图2-8 算法4

不难理解为什么时间的界是正确的，但是要明白为什么算法是正确可行的会费些精力。为了分析原因，注意，像算法1和算法2一样， $j$ 代表当前序列的终点，而 $i$ 代表当前序列的起点。碰巧的是，如果我们不需要知道最佳的子序列在哪里，那么 $i$ 的使用可以从程序中优化掉，但在设计算法的时候还是假设 $i$ 是需要的，而且我们想要改进算法2。一个结论是，如果 $a[i]$ 是负的，那么它不可能代表最优序列的起点，因为任何包含 $a[i]$ 的作为起点的子序列都可以通过用 $a[i+1]$ 作起点而得到改进。类似地，任何负的子序列不可能是最优子序列的前缀（原理相同）。如果在内循环中我们检测到从 $a[i]$ 到 $a[j]$ 的子序列是负的，那么我们可以推进 $i$ 。关键的结论是，我们不仅能够把 $i$ 推进到 $i+1$ ，而且我们实际上还可以把它一直推进到 $j+1$ 。为了看清楚这一点，令 $p$ 为 $i+1$ 和 $j$ 之间的任一下标。开始于下标 $p$ 的任意子序列都不大于在下标 $i$ 开始并包含从 $a[i]$ 到 $a[p-1]$ 的子序列的对应子序列，因为后面这个子序列不是负的（ $j$ 是使得从下标 $i$ 开始成为负值的第一个下标）。因此，把 $i$ 推进到 $j+1$ 是安全：我们不会错过最优解。

这个算法是许多聪明算法的典型：运行时间是明显的，但正确性则不明显。对于这些算法，正式的正确性证明（比上面的分析更正式）几乎总是需要的；然而，即便如此，许多人仍然不相信。此外，许多这类算法需要更有技巧的编程，这导致更长的开发过程。不过当这些算法正常工作时，它们运行得很快，而我们将它们和一个低效（但容易实现）的蛮力算法通过小规模输入进行比较，可以测试到大部分的程序原理。

该算法的一个附带的优点是，它只对数据进行一次扫描，一旦 $a[i]$ 被读入并被处理，它就不再需要被记忆。因此，如果数组在磁盘或磁带上，它就可以被顺序读入，在主存中不必存储数组的任何部分。不仅如此，在任意时刻，算法都能对它已经读入的数据给出子序列问题的正确答案（其他算法不具有这个特性）。具有这种特性的算法叫作**联机算法**（on-line algorithm）。仅需要常量空间并以线性时间运行的联机算法几乎是完美的算法。

### 2.4.4 运行时间中的对数

分析算法最混乱的方面大概集中在对数上面。我们已经看到，某些分治算法将以 $O(M \log N)$ 时间运行。除分治算法外，对数最常出现的规律可概括为下列一般法则：如果一个算法用常数时间（ $O(1)$ ）将问题的大小削减为其一部分（通常是 $1/2$ ），那么该算法就是 $O(\log N)$ 的。另一方面，如果使用常数时间只是把问题减少一个常数的数量（如将问题减少1），那么这种算法就是 $O(N)$ 的。

显然，只有一些特殊种类的问题才能够呈 $O(\log N)$ 型。例如，若输入 $N$ 个数，则一个算法只要把这些数读入就必须耗费 $\Omega(N)$ 的时间量。因此，当我们谈到这类问题的 $O(\log N)$ 算法时，通常都是假设输入数据已经提前读入。我们提供具有对数特点的三个例子。

#### 1. 二分搜索

第一个例子通常叫作二分搜索（binary search）。

**二分搜索** 给定一个整数 $X$ 和整数 $A_0, A_1, \dots, A_{N-1}$ ，后者已经预先排序并在内存中，求下标 $i$ 使得 $A_i = X$ ，如果 $X$ 不在数据中，则返回 $i = -1$ 。

明显的解法是从左到右扫描数据，其运行花费线性时间。然而，这个算法没有用到该表已经排序的事实，这就使得算法很可能不是最好的。一个好的策略是验证 $X$ 是否是居中的元素。如果是，则答案就找到了。如果 $X$ 小于居中元素，那么我们可以应用同样的策略于居中元素左边已排序的子序列；同理，如果 $X$ 大于居中元素，那么我们检查数据的右半部分。（同样，也存在可能会终止的情况。）图2-9列出了二分搜索的程序（其答案为mid）。图2-9中的程序反映了C++语言数组下标从0开始的惯例。

58

```

1  /**
2   * Performs the standard binary search using two comparisons per level.
3   * Returns index where item is found or -1 if not found.
4   */
5  template <typename Comparable>
6  int binarySearch( const vector<Comparable> & a, const Comparable & x )
7  {
8      int low = 0, high = a.size( ) - 1;
9
10     while( low <= high )
11     {
12         int mid = ( low + high ) / 2;
13
14         if( a[ mid ] < x )
15             low = mid + 1;
16         else if( a[ mid ] > x )
17             high = mid - 1;
18         else
19             return mid;    // Found
20     }
21     return NOT_FOUND;    // NOT_FOUND is defined as -1
22 }

```

图2-9 二分搜索

显然，每次迭代在循环内的所有工作花费 $O(1)$ 的时间，因此分析需要确定循环的次数。循环从 $high - low = N - 1$ 开始并在 $high - low \geq -1$ 结束。每次循环后 $high - low$ 的值至少将该次循环前的值折半；于是，循环的次数最多为 $\lceil \log(N - 1) \rceil + 2$ 。（例如，若 $high - low = 128$ ，则在各次迭代后 $high - low$ 的最大值是64, 32, 16, 8, 4, 2, 1, 0, -1。）因此，运行时间是 $O(\log N)$ 。等价地，我

们也可以写出运行时间的递归公式，不过，当我们理解实际在做什么以及为什么的原理时，这种强行写公式的做法通常没有必要。

二分搜索可以看作是我们的第一个数据结构实现方法，它提供了在 $O(\log N)$ 时间内的contains操作，但是所有其他操作（特别是insert操作）均需要 $O(N)$ 时间。在数据是稳定（即不允许插入操作和删除操作）的应用中，这可能是非常有用的。此时输入数据需要一次排序，但是此后的访问会很快。有个例子是一个程序，它需要保留（产生于化学和物理中的）元素周期表的信息。这个表是相对稳定的，因为很少会加进新的元素。元素名可以始终是排序的。由于只有大约110种元素，因此找到一个元素最多需要存取8次。要是执行顺序查找就会需要多得多的访问次数。

59

## 2. 欧几里得算法

第二个例子是计算最大公因数的欧几里得算法。两个整数的最大公因数（gcd）是同时整除二者的最大整数。于是， $\text{gcd}(50, 15) = 5$ 。图2-10中的算法计算 $\text{gcd}(M, N)$ ，假设 $M \geq N$ 。（如果 $N > M$ ，则循环的第一次迭代将它们互相交换）。

```

1 long gcd( long m, long n )
2 {
3     while( n != 0 )
4     {
5         long rem = m % n;
6         m = n;
7         n = rem;
8     }
9     return m;
10 }
```

图2-10 欧几里得算法

算法通过连续计算余数直到余数是0为止，最后的非零余数就是最大公因数。因此，如果 $M=1989$ 和 $N=1590$ ，则余数序列是399, 393, 6, 3, 0。从而， $\text{gcd}(1989, 1590) = 3$ 。正如例子所表明，这是一个快速算法。

如前所述，估计算法的整个运行时间依赖于确定余数序列究竟有多长。虽然 $\log N$ 看似是理想中的答案，但是根本看不出余数的值按照常数因子递减的必然性，因为我们看到，例中的余数从399仅仅降到393。事实上，在一次迭代中余数并不按照一个常数因子递减。然而，我们可以证明，在两次迭代以后，余数最多是原始值的一半。这就证明了，迭代次数至多是 $2 \log N = O(\log N)$ 从而得到运行时间。这个证明并不难，因此我们将它包含在这里，它可从下列定理直接推出。

**定理2.1** 如果 $M > N$ ，则 $M \bmod N < M/2$ 。

**证明** 存在两种情形。如果 $N \leq M/2$ ，则由于余数小于 $N$ ，故定理在这种情形下成立。另一种情形是 $N > M/2$ 。但是此时 $M$ 仅含有一个 $N$ 从而余数为 $M - N < M/2$ ，定理得证。 ■

从上面的例子来看， $2 \log N$ 大约为20，而我们仅进行了7次运算，因此有人会怀疑这是不是可能的最好的界。事实上，这个常数在最坏的情况下（如 $M$ 和 $N$ 是两个相邻的斐波那契数时就是这种情况）还可以稍微改进成 $1.44 \log N$ 。欧几里得算法在平均情况下的性能需要大量篇幅的高度复杂的数学分析，其迭代的平均次数约为 $(12 \ln 2 \ln N) / \pi^2 + 1.47$ 。

60

## 3. 幂运算

我们在本节的最后一个例子是处理一个整数的幂（它还是一个整数）。由取幂运算得到的数一般都是相当大的，因此，我们只能在假设有一台机器能够存储这样一些大整数（或有一个编译



程序能够模拟它)的情况下进行分析。我们将用乘法的次数作为运行时间的度量。

计算 $X^N$ 的明显的算法是使用 $N-1$ 次乘法自乘。使用递归算法会更好。 $N \leq 1$ 是递归的基准情形。如果 $N$ 是偶数,我们有 $X^N = X^{N/2} \cdot X^{N/2}$ ,如果 $N$ 是奇数,则 $X^N = X^{(N-1)/2} \cdot X^{(N-1)/2} \cdot X$ 。

例如,为了计算 $X^{62}$ ,算法将如下进行,它只用到9次乘法:

$$X^3 = (X^2) X, X^7 = (X^3)^2 X, X^{15} = (X^7)^2 X, X^{31} = (X^{15})^2 X, X^{62} = (X^{31})^2$$

显然,所需要的乘法次数最多是 $2\log N$ ,因为把问题分半最多需要两次乘法(如果 $N$ 是奇数)。这里,我们又可写出一个递推公式并将其解出。简单的直觉避免了盲目的强行处理。

图2-11是该思想的实现。有时候看一看程序能够进行多大的调整而不影响其正确性是很有趣的。在图2-11中,第5行到第6行实际上不是必需的,因为如果 $N$ 是1,那么第10行将做同样的事情。第10行还可以写成

```
10 return pow (x, n - 1) * x;
```

而不影响程序的正确性。事实上,程序仍将以 $O(\log n)$ 运行,因为乘法的序列同以前一样。不过,虽然看起来似乎都是正确的,但下面所有对第8行的修改都是不可取的:

```
8a return pow ( pow ( x, 2), n / 2);
8b return pow ( pow (x, n / 2), 2);
8c return pow (x, n / 2 ) * pow (x, n / 2 );
```

61

8a和8b两行都是不正确的,因为当 $N$ 是2的时候,递归调用pow中有一个是以2作为第2个参数。这样,程序产生一个无限循环,将不能往下进行(最终导致程序非正常终止)。

使用8c行影响程序的效率,因为此时有两个大小为 $N/2$ 的递归调用而不是一个。分析指出,其运行时间不再是 $O(\log N)$ 。我们把确定新的运行时间作为练习留给读者。

```
1 long pow( long x, int n )
2 {
3     if( n == 0 )
4         return 1;
5     if( n == 1 )
6         return x;
7     if( isEven( n ) )
8         return pow( x * x, n / 2 );
9     else
10        return pow( x * x, n / 2 ) * x;
11 }
```

图2-11 高效率的幂运算

### 2.4.5 检验你的分析

一旦分析进行过后,则需要看一看答案是否正确,是否尽可能地好。一种方法是编程并比较实际观察到的运行时间是否与通过分析所描述的运行时间相匹配。如果 $N$ 扩大一倍,则线性程序的运行时间乘以系数2,二次程序的运行时间乘以系数4,而三次程序的运行时间则乘以系数8。以对数时间运行的程序当 $N$ 增加一倍时其运行时间只是多加一个常数,而以 $O(M\log N)$ 运行的程序则花费比在相同环境下运行时间的两倍稍多一些的时间。如果低阶项的系数相对较大,并且 $N$ 又不是足够大,那么运行时间的这种增加量很难观察清楚。例如,对于最大子序列和问题,当从 $N=10$ 增到 $N=100$ 时,运行时间的变化就是一个例子。单纯凭经验区分线性程序和 $O(M\log N)$ 程序是非常困难的。

验证一个程序是否是 $O(f(N))$ 的另一个常用的技巧是对 $N$ 的某个范围计算比值 $T(N)/f(N)$ (通常

用2的倍数隔开), 其中 $T(N)$ 是凭经验观察到的运行时间。如果 $f(N)$ 是运行时间的理想近似, 那么所算出的值收敛于一个正常数。如果 $f(N)$ 估计过大, 则算出的值收敛于0。如果 $f(N)$ 估计过低从而 $O(f(N))$ 是错的, 那么算出的值发散。

作为一个例子, 图2-12中的程序段计算两个随机选取、小于或等于 $N$ 的互异正整数互素的概率。(当 $N$ 增大时, 结果将趋向于 $6/\pi^2$ 。)

62

```

1 double probRelPrime( int n )
2 {
3     int rel = 0, tot = 0;
4
5     for( int i = 1; i <= n; i++ )
6         for( int j = i + 1; j <= n; j++ )
7             {
8                 tot++;
9                 if( gcd( i, j ) == 1 )
10                     rel++;
11             }
12
13     return (double) rel / tot;
14 }

```

图2-12 估计两个随机数互素的概率

读者应该能够立即对这个程序做出分析。图2-13显示实际观察到的该例程在一台具体计算机上的运行时间。该图表指出, 表中的最后一列是最有可能的, 因此所得出的这个分析很可能正确。注意, 在 $O(N^2)$ 和 $O(N^2 \log N)$ 之间没有多大差别, 因为对数增长得很慢。

N	CPU 时间(T)	$T/N^2$	$T/N^3$	$T/(N^2 \log N)$
100	022	0.002200	0.000022000	0.0004777
200	056	0.001400	0.000007000	0.0002642
300	118	0.001311	0.000004370	0.0002299
400	207	0.001294	0.000003234	0.0002159
500	318	0.001272	0.000002544	0.0002047
600	466	0.001294	0.000002157	0.0002024
700	644	0.001314	0.000001877	0.0002006
800	846	0.001322	0.000001652	0.0001977
900	1 086	0.001341	0.000001490	0.0001971
1 000	1 362	0.001362	0.000001362	0.0001972
1 500	3 240	0.001440	0.000000960	0.0001969
2 000	5 949	0.001482	0.000000740	0.0001947
4 000	25 720	0.001608	0.000000402	0.0001938

图2-13 对上述例程的经验运行时间

#### 2.4.6 分析结果的准确性

经验指出, 有时分析会估计过大。如果发生这种情况, 那么或者需要分析得更细(一般通过机敏的观察), 或者可能是平均运行时间显著小于最坏情形的运行时间而又不可能对所得的界再加以改进。对于许多复杂的算法, 最坏的界通过某个不良输入是可以达到的, 但在实践中它通常是估计过大的。遗憾的是, 对于大多数这种问题, 平均情形的分析是极其复杂的(在许多情形下还是未解决的), 而最坏情形的界尽管过分悲观但却是最好的已知分析结果。

## 小结

63

本章对如何分析程序的复杂性给出了一些提示。遗憾的是，它并不是完善的分析指南。简单的程序通常给出简单的分析，但是情况也并不总是如此。作为一个例子，在本书稍后我们将看到一个排序算法（谢尔排序，第7章）和一个维持不相交集的算法（第8章），它们大约都需要20行程序代码。谢尔排序（Shellsort）的分析仍然不完善，而不相交算法的分析非常困难，需要许多页错综复杂的计算。不过，我们在这里遇到的大部分分析都是简单的，它们涉及对循环的计数。

一类有趣的分析是下界分析，我们尚未接触到。在第7章我们将看到这方面的一个例子：证明任何仅通过使用比较来进行排序的算法在最坏的情形下只需要 $\Omega(M \log N)$ 次比较。下界的证明一般是最困难的，因为它们不只适用求解某个问题的一个算法而是适用求解该问题的一类算法。

在本章结束前，我们指出此处描述的某些算法在实际生活中的应用。gcd算法和求幂算法应用在密码学中。特别地，400位数字的数自乘至一个大的幂次（通常为另一个400位数字的数）而在每乘一次后只有低400位左右的数字保留下来。由于这种计算需要处理200位数字的数，因此效率显然是非常重要的。求幂运算的直接相乘会需要大约 $10^{400}$ 次乘法，而上面描述的算法在最坏情形下只需要大约2600次乘法。

## 练习

- 2.1 按增长率排列下列函数： $N$ ,  $\sqrt{N}$ ,  $N^{1.5}$ ,  $N^2$ ,  $M \log N$ ,  $M \log \log N$ ,  $M \log^2 N$ ,  $M \log(N^2)$ ,  $2/N$ ,  $2^N$ ,  $2^{N/2}$ ,  $37$ ,  $N^2 \log N$ ,  $N^3$ 。指出哪些函数以相同的增长率增长。
- 2.2 设 $T_1(N) = O(f(N))$ 和 $T_2(N) = O(f(N))$ 。下列等式哪些成立？
  - a.  $T_1(N) + T_2(N) = O(f(N))$
  - b.  $T_1(N) - T_2(N) = o(f(N))$
  - c.  $\frac{T_1(N)}{T_2(N)} = O(1)$
  - d.  $T_1(N) = O(T_2(N))$
- 2.3 哪个函数增长得更快： $M \log N$ ，还是 $N^{1+\varepsilon/\sqrt{\log N}}$  ( $\varepsilon > 0$ )？
- 2.4 证明：对任意常数 $k$ ,  $\log^k N = o(N)$ 。
- 2.5 求两个函数 $f(N)$ 和 $g(N)$ 使得既不是 $f(N) = O(g(N))$ ，也不是 $g(N) = O(f(N))$ 。
- 2.6 在最近的一次法庭审理案件中，一位法官因藐视法庭罪传讯一个市民并命令第一天交纳罚金2美元，以后每天的罚金都要将上一天的罚金数额平方，直到该市民服从该法官的命令为止（即，罚金上升如下：2美元，4美元，16美元，256美元，65 536美元，……）。
  - a. 在第 $N$ 天罚金将是多少？
  - b. 使罚金达到 $D$ 美元需要多少天（使用大 $O$ ）？
- 2.7 对于下列六个程序片段中的每一个：
  - a. 给出运行时间分析（使用大 $O$ ）。
  - b. 用你选择的程序语言编程，并对 $N$ 的若干具体值给出运行时间。
  - c. 用实际的运行时间与你所做的分析进行比较。
    - (1) 

```
sum = 0;
for(i = 0; i < n; i++)
    sum++;
```
    - (2) 

```
sum = 0;
```

64



```

    for(i = 0; i < n; i++)
        for(j=0; j < n; j++)
            sum++;
(3) sum = 0;
    for(i = 0; i < n; i++)
        for(j=0; j < n * n; j++)
            sum++;
(4) sum = 0;
    for(i = 0; i < n; i++)
        for(j=0; j < i; j++)
            sum++;
(5) sum = 0;
    for(i = 0; i < n; i++)
        for(j=0; j < i * i; j++)
            for( k = 0; k < j; k++)
                sum++;
(6) sum = 0;
    for(i = 1; i < n; i++)
        for(j = 1; j < i * i; j++)
            if( j % i == 0)
                for( k = 0; k < j; k++)
                    sum++;

```

2.8 假设需要生成前 $N$ 个自然数的一个随机置换。例如， $\{4, 3, 1, 5, 2\}$ 和 $\{3, 1, 4, 2, 5\}$ 就是合法的置换，但 $\{5, 4, 1, 2, 1\}$ 则不是，因为数1出现两次而数3却没有。这个程序常常用于模拟一些算法。我们假设存在一个随机数生成器 $r$ ，它有方法 $\text{randInt}(i, j)$ ，以相同的概率生成 $i$ 和 $j$ 之间的一个整数。下面是三个算法：

- (1) 如下填入从 $a[0]$ 到 $a[N-1]$ 的数组 $a$ ：为了填入 $a[i]$ ，生成随机数直到它不同于已经生成的 $a[0], a[1], \dots, a[i-1]$ 时再将其填入 $a[i]$ 。
- (2) 同算法(1)，但是要使用一个附加的数组，称之为 $\text{used}$ 数组。当一个随机数 $\text{ran}$ 最初被放入数组 $a$ 的时候，设置 $\text{used}[\text{ran}] = \text{true}$ 。这就是说，当用一个随机数填入 $a[i]$ 时，可以用一步来测试是否该随机数已经被使用，而不是像第一个算法那样（可能）用 $i$ 步测试。
- (3) 填写该数组使得 $a[i] = i + 1$ 。然后

```

for(i = 1; i < n; i++)
    swap (a[ i ], a [ randInt ( 0, i ) ] );

```

- a. 证明这三个算法都生成合法的置换，并且所有的置换都是等可能的。
- b. 对每一个算法给出你能够得到的尽可能准确的期望运行时间分析（用大 $O$ ）。
- c. 分别写出程序来执行每个算法10次，得出一个好的平均值。对 $N = 250, 500, 1000, 2000$ 运行程序(1)；对 $N = 25\,000, 50\,000, 100\,000, 200\,000, 400\,000, 800\,000$ 运行程序(2)；对 $N = 100\,000, 200\,000, 400\,000, 800\,000, 1\,600\,000, 3\,200\,000, 6\,400\,000$ 运行程序(3)。
- d. 将实际的运行时间与你的分析进行比较。
- e. 每个算法的最坏情形的运行时间是什么？

65

2.9 用运行时间的估计值完成图2-2中的表，当时这些估计值太长无法模拟。插入上述三个算法的运行时间并估计计算100万个数的最大子序列和所需要的时间。你得出哪些假设？

2.10 对于手工进行计算所使用的典型算法，确定做下列计算的运行时间：

- a. 将两个 $N$ 位数字的整数相加。
- b. 将两个 $N$ 位数字的整数相乘。
- c. 将两个 $N$ 位数字的整数相除。

2.11 一个算法对于大小为100的输入花费0.5 ms。如果运行时间如下，则解决输入大小为500的问题需要花费多长的时间（设低阶项可以忽略）。

- a. 是线性的。

- b. 为 $O(M\log N)$ 。  
 c. 是二次的。  
 d. 是三次的。
- 2.12 一个算法对于大小为100的输入花费0.5 ms。如果运行时间如下，则用1 min可以解决多大的问题（设低阶项可以忽略）。
- a. 是线性的。  
 b. 为 $O(M\log N)$ 。  
 c. 是二次的。  
 d. 是三次的。
- 2.13 计算 $f(x) = \sum_{i=0}^N a_i x^i$  需要多少时间？
- a. 用简单的例程执行求幂运算。  
 b. 使用2.4.4节的例程计算。
- 2.14 考虑下述算法（称为Horner法则）计算 $f(x) = \sum_{i=0}^N a_i x^i$  的值：
- ```
poly = 0;
for( i = n; i >= 0; i--)
    poly = x * poly + a[i];
```
- a. 对 $x=3$ ,  $f(x) = 4x^4 + 8x^3 + x + 2$ , 指出该算法的各步是如何进行的。  
 b. 解释该算法为什么能够解决这个问题。  
 c. 该算法的运行时间是多少？
- 2.15 给出一个有效的算法来确定在整数 $A_1 < A_2 < A_3 < \dots < A_N$ 的数组中是否存在整数 $i$ 使得 $A_i = i$ 。你的算法的运行时间是多少？
- 2.16 基于下列各式编写替代的gcd算法（其中 $a > b$ ）。
- a.  $\gcd(a, b) = 2\gcd(a/2, b/2)$ , 若 $a$ 和 $b$ 均为偶数。  
 b.  $\gcd(a, b) = \gcd(a/2, b)$ , 若 $a$ 为偶数,  $b$ 为奇数。  
 c.  $\gcd(a, b) = \gcd(a, b/2)$ , 若 $a$ 为奇数,  $b$ 为偶数。  
 d.  $\gcd(a, b) = \gcd((a+b)/2, (a-b)/2)$ , 若 $a$ 和 $b$ 均为奇数。
- 2.17 给出有效的算法（及其运行时间分析）来：
- a. 求最小子序列和。  
 \*b. 求最小的正子序列和。  
 \*c. 求最大子序列乘积。
- 2.18 数值分析中一个重要的问题是对某个任意的函数 $f$ 找出方程 $f(X) = 0$ 的一个解。如果该函数是连续的并有两个点 $low$ 和 $high$ 使得 $f(low)$ 和 $f(high)$ 符号相反，那么在 $low$ 和 $high$ 之间必然存在一个根，并且这个根可以通过二分搜索求得。写出一个函数，以 $f$ 、 $low$ 和 $high$ 为参数，并且解出一个零点。为保证程序正常终止，你必须做什么？
- 2.19 课文中最大相连子序列和算法均不给出具体的序列的任何指示。将这些算法修改使得它们以单个对象的形式返回最大子序列的值以及具体的序列的下标。
- 2.20
- a. 编写一个程序来确定正整数 $N$ 是否是素数。  
 b. 你的程序在最坏情形下的运行时间是多少（用 $N$ 表示）？（你应该能够写出 $O(\sqrt{N})$ 的算法程序。）  
 c. 令 $B$ 等于 $N$ 的二进制表示法中的位数。 $B$ 的值是什么？  
 d. 你的程序在最坏情形下的运行时间是什么（用 $B$ 表示）？  
 e. 比较确定一个20位的数是否是素数和确定一个40位的数是否是素数的运行时间。  
 f. 用 $N$ 或 $B$ 给出运行时间更合理吗？为什么？
- \*2.21 厄拉多塞（Erasthenes）筛是一种用于计算小于 $N$ 的所有素数的方法。我们从制作整数2到 $N$ 的表

开始。我们找出最小的未被删除的整数 $i$ ，打印 $i$ ，然后删除 $i, 2i, 3i, \dots$ 。当 $i > \sqrt{N}$ 时，算法终止。该算法的运行时间是多少？

- 2.22 证明 $X^{62}$ 可以只用8次乘法算出。
- 2.23 不用递归，写出快速求幂的程序。
- 2.24 给出用于快速求幂运算中的乘法次数的精确计数。（提示：考虑 $N$ 的二进制表示。）
- 2.25 程序 $A$ 和 $B$ 经分析发现其最坏情形运行时间分别不大于 $150N \log_2 N$ 和 $N^2$ 。如果可能，请回答下列问题：

- $N$ 值很大时（ $N > 10\,000$ ），哪一个程序的运行时间有更好的保障？
- $N$ 值很小时（ $N < 100$ ），哪一个程序的运行时间更少？
- 对于 $N = 1000$ ，哪一个程序平均运行得更快？
- 对于所有可能的输入，程序 $B$ 是否总能够比程序 $A$ 运行得更快？

- 2.26 大小为 $N$ 的数组 $A$ ，其主元素是一个出现超过 $N/2$ 次的元素（从而这样的元素最多有一个）。例如，数组

3, 3, 4, 2, 4, 4, 2, 4, 4

有一个主元素4，而数组

3, 3, 4, 2, 4, 4, 2, 4

没有主元素。如果没有主元素，那么你的程序应该指出来。下面是求解该问题的一个算法的概要：首先，找出主元素的一个候选元（这是较困难的部分）。这个候选元是唯一有可能是主元素的元素。第二步确定是否该候选元实际上就是主元素。这正好是对数组的顺序搜索。为找出数组 $A$ 的一个候选元，构造第二个数组 $B$ 。比较 $A_1$ 和 $A_2$ ，如果它们相等，则取其中之一加到数组 $B$ 中；否则什么也不做。然后比较 $A_3$ 和 $A_4$ ，同样地，如果它们相等，则取其中之一加到 $B$ 中；否则什么也不做。以该方式继续下去直到读完整个数组。然后，递归地寻找数组 $B$ 中的候选元；它也是 $A$ 的候选元。（为什么？）

- 递归如何终止？
  - 当 $N$ 是奇数时的情形如何处理？
  - 该算法的运行时间是多少？
  - 我们如何避免使用附加数组 $B$ ？
  - 编写一个程序，求解主元素。
- 2.27 输入是一个 $N \times N$ 数字矩阵并且已经读入内存。每一行均从左到右增加。每一列则从上到下增加。给出一个 $O(N)$ 最坏情形算法以决定是否数 $X$ 在该矩阵中。
- 2.28 设计有效的算法使用正数的数组 $a$ 以确定：
- $a[j] + a[i]$ 的最大值，其中 $j \geq i$ 。
  - $a[j] - a[i]$ 的最大值，其中 $j \geq i$ 。
  - $a[j] * a[i]$ 的最大值，其中 $j \geq i$ 。
  - $a[j] / a[i]$ 的最大值，其中 $j \geq i$ 。
- \*2.29 在我们的计算机模型中为什么假设整数具有固定长度是很重要的？
- 2.30 考虑第1章中描述的字谜游戏问题。假设我们固定最长单词的大小为10个字符。
- 设 $R$ 、 $C$ 和 $W$ 分别表示字谜游戏中的行数、列数和单词个数，那么在第1章所描述的算法用 $R$ 、 $C$ 和 $W$ 表示的运行时间是多少？
  - 设单词表是预先排序过的。指出如何使用二分搜索得到一个具有少得多的运行时间的算法。
- 2.31 设在二分搜索程序的第15行的语句是 $low = mid$ 而不是 $low = mid + 1$ 。这个程序还能正确运行吗？
- 2.32 实现二分搜索使得在每次迭代中只执行一次二路比较。



2.33 设算法3（图2-7）的第15行和第16行由

```
15 int maxLeftSum = maxSubSum(a, left, center - 1);  
16 int maxRightSum = maxSubSum(a, center, right);
```

代替，这个程序还能正确运行吗？

\*2.34 立方最大子序列和算法的内循环执行 $N(N+1)(N+2)/6$ 次最内层代码的迭代。相应的二次算法执行 $N(N+1)/2$ 次迭代。而线性算法执行 $N$ 次迭代。哪种模式是显然的？你能给出这种现象的组合学解释吗？

---

## 参考文献

---

算法的运行时间分析最初因Knuth在其三卷本丛书[5]、[6]和[7]中使用而流行。gcd算法的分析出现在[6]中。这方面的另一本早期著作见于[1]。

大O、大 $\Omega$ 、大 $\Theta$ 以及小o记号由Knuth在[8]中提倡。但是对于这些记号尚无统一的规定，特别是在使用 $\Theta()$ 时。许多人更愿意使用 $O()$ ，虽然它表达的精度要差得多。此外，当需要用到 $\Omega()$ 时，迫不得已还用 $O()$ 表示下界。

最大子序列和问题出自[3]。丛书[2]、[3]和[4]指出如何优化程序以提高其运行速度。

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. J. L. Bentley, *Writing Efficient Programs*, Prentice Hall, Englewood Cliffs, N.J., 1982.
3. J. L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, Mass., 1986.
4. J. L. Bentley, *More Programming Pearls*, Addison-Wesley, Reading, Mass., 1988.
5. D. E. Knuth, *The Art of Computer Programming, Vol 1: Fundamental Algorithms*, 3rd ed., Addison-Wesley, Reading, Mass., 1997.
6. D. E. Knuth, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, Reading, Mass., 1998.
7. D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching*, 2nd ed., Addison-Wesley, Reading, Mass., 1998.
8. D. E. Knuth, "Big Omicron and Big Omega and Big Theta," *ACM SIGACT News*, 8 (1976), 18-23.

## 表、栈和队列

**本**章讨论最简单和最基本的三种数据结构。实际上，每一个有意义的程序都将明晰地至少使用一个这样的数据结构，而栈则在程序中总是要间接地用到，而不管你在程序中是否进行了声明。本章将：

- 介绍抽象数据类型（ADT）的概念。
- 阐述如何对表进行高效的操作。
- 介绍栈ADT及其在实现递归方面的应用。
- 介绍队列ADT及其在操作系统和算法设计中的应用。

在本章中，给出了两个库类vector和list的重要子集的实现代码。

### 3.1 抽象数据类型（ADT）

**抽象数据类型**（abstract data type, ADT）是带有一组操作的一些对象的集合。抽象数据类型是数学的抽象；在ADT的定义中根本没有提到这组操作是如何实现的。诸如表、集合、图以及它们各自的操作一起形成的这些对象都可以看做是抽象数据类型，就像整数、实数、布尔数都是数据类型一样。整数、实数和布尔数各自都有与之相关的操作，而抽象数据类型也是如此。对于集合ADT，可以有像加（add）、删除（remove）、大小（size）以及包含（contains）这样一些操作。当然，也可以只要两种操作：并（union）和查找（find），这两种操作又在该集合上定义了一种不同的ADT。

C++的类也考虑到ADT的实现，不过适当地隐藏了实现的细节。这样，程序中需要对ADT实施操作的任何其他部分可以通过调用适当的方法来进行。如果由于某种原因需要改变实现的细节，那么通过仅仅改变执行这些ADT操作的例程应该是很容易做到的。理想情况下，这种改变对于程序的其余部分是完全透明的。

对于每种ADT并不存在什么法则来告诉我们必须要有哪些操作；这是一个设计决策。错误处理和结构调整（在适当的地方）一般也取决于程序的设计者。本章中将要讨论的三种数据结构是ADT的最基本的例子，我们将会看到它们中的每一种是如何以多种方法实现的。不过，如果这些实现是正确的，那么使用它们的程序就没有必要知道使用的是哪个实现。

71

### 3.2 表ADT

我们将处理形如 $A_0, A_2, A_3, \dots, A_{N-1}$ 的一般的表。这个表的大小是 $N$ 。我们将称大小为0的表为**空表**（empty list）。

对于除空表外的任何表，我们说 $A_i$ 后继 $A_{i-1}$ （或继 $A_{i-1}$ 之后）并称 $A_{i-1}$ （ $i < N$ ）前驱 $A_i$ （ $i > 1$ ）。



表中的第一个元素是 $A_0$ ，而最后一个元素是 $A_{N-1}$ 。我们将不定义 $A_0$ 的前驱元，也不定义 $A_{N-1}$ 的后继元。元素 $A_i$ 在表中的位置（position）为 $i$ 。为了简单起见，我们在讨论中将假设表中的元素是整数，但一般说来任意的复杂元素也是允许的（而且很容易由类模板来处理）。

与这些“定义”相关的是我们要在表ADT上进行操作的集合。`printList`和`makeEmpty`是常用的操作，其功能是显而易见的；`find`返回项首次出现的位置；`insert`和`remove`一般是从表的某个位置插入和删除一些元素；而`findKth`则返回某个位置上（作为参数而被指定）的元素。如果34, 12, 52, 16, 12是一个表，则`find(52)`会返回2；`insert(x, 2)`可把表变成34, 12, x, 52, 16, 12（如果我们在给定位置插入的话）；而`remove(52)`则将该表变为34, 12, x, 16, 12。

当然，一个函数的功能怎样才算恰当，完全要由程序设计者来确定。这类似于对特殊情况的处理（例如，上述`find(1)`返回什么？）。我们还可以添加一些运算，比如`next`和`previous`，它们会取一个位置作为参数并分别返回其后继元和前驱元的位置。

### 3.2.1 表的简单数组实现

对表的所有操作都可以使用数组来实现。虽然数组是静态分配的，但是内部存储数组的`vector`类允许在需要的时候将数组的大小增加一倍。这解决了使用数组的最严重的问题。也就是在使用数组时需要表的最大的最大值进行估计的问题。这个估计现在就不再需要了。

数组实现使得`printList`以线性时间执行，而`findKth`则花费常数时间，这是最好的结果了。然而，插入和删除的花费却有可能是昂贵的，这取决于插入和删除发生的位置。在最坏的情况下，在位置0（换句话说是在表的前面）插入需要将整个数组后移一个位置以空出空间来；而删除第一个元素则需要将表中的所有元素前移一个位置，因此这两种操作的最坏情况为 $O(N)$ 。平均来看，这两种运算都需要移动表的一半的元素，因此仍然需要线性时间。另一方面，如果所有的操作都发生在表的末尾，就不需要移动任何元素，那么添加和删除的操作都花费 $O(1)$ 的时间。

在许多情况下，表是通过在末尾插入元素来建立的，之后只有数组访问（例如`findKth`操作）发生。在这种情况下，数组是适合的。然而，如果插入和删除在整个表中都发生，特别是在表的前端发生的话，数组就不是一个好选择了。下一节讨论另一种选择：链表。

72

### 3.2.2 简单链表

为了避免插入和删除的线性开销，我们需要允许表可以不连续存储，否则表的部分或全部就需要整体移动。图3-1表达了链表（linked list）的一般思想。

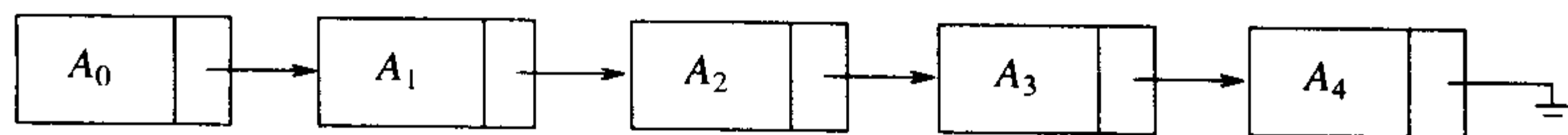


图3-1 一个链表

链表由一系列不必在内存中相连的结点组成。每一个结点均含有表元素和到包含该元素后继元的结点的链（link）。我们称之为`next`链。最后一个单元的`next`链指向`NULL`。

为了执行`printList()`或`find(x)`，我们只要从表的第一个结点开始然后用`next`链遍历该表即可。与数组实现一样，这种操作显然是花费线性时间的，但是这个常数可能比用数组实现时要大。`findKth`操作不如数组实现时的效率高；`findKth(i)`花费 $O(i)$ 的时间并以明显的方式遍历链表而完成。在实践中这个界是保守的，因为调用`findKth`常常是以（按 $i$ ）排序的方式进行。例如，



`findKth(2)`、`findKth(3)`、`findKth(4)` 以及 `findKth(6)` 可通过对表的一次扫描同时实现。

`remove` 方法可以通过修改一个 `next` 引用来实现。图3-2给出在原表中删除第二个元素的结果。

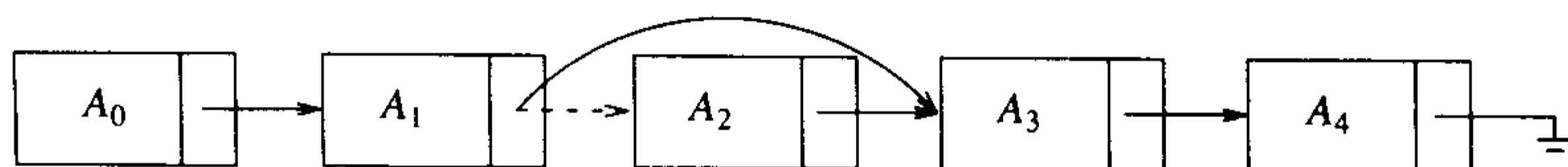


图3-2 从链表中删除

`insert` 方法需要使用 `new` 操作符从系统取得一个新结点，此后执行两次引用调整。其一般想法在图3-3中给出，其中的虚线表示原来的指针。

73

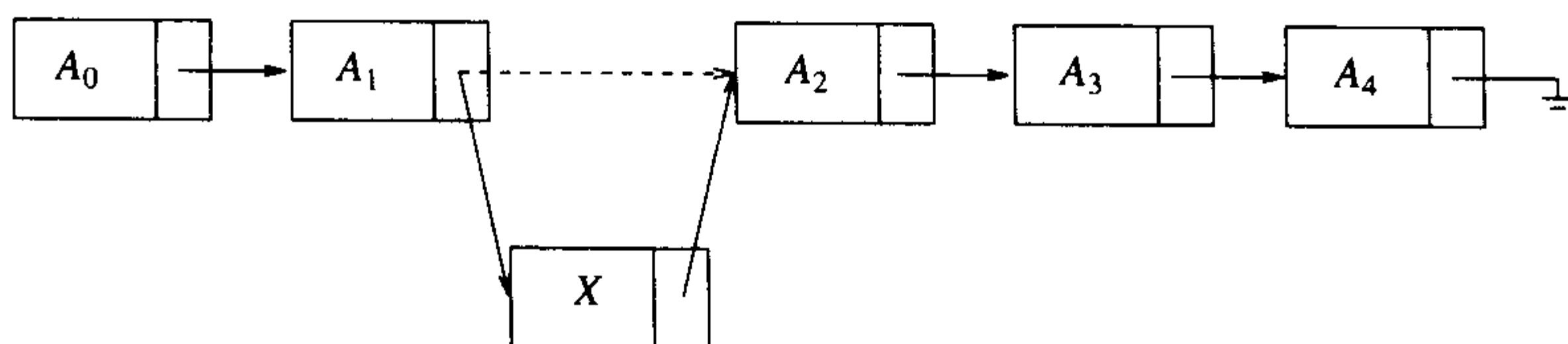


图3-3 向链表插入

一般地，正如我们所看到的，如果知道发生变化的位置，从链表中插入和删除一个元素不需要移动很多的元素，而只是对结点链接进行固定的几个改变。

假设链接到链表前面的链接存在，那么，添加或删除第一项的特殊情况对应常量的时间。至于在链表的末尾添加（在最后一项后添加新项）的特殊情况，如果链接到最后项的链接存在的话，也是消耗常量的时间。这样，典型的链表保持至表的两端的链接。删除最后一项有点麻烦，因为必须找到最后项前面的项，更改其 `next` 链接到 `NULL`，然后更新这个保持为最后一项的链接。在经典的链表里每个结点存储指向下一结点的链接，但是没有提供关于上一个结点的任何信息。

显而易见的想法是构造第三个链接来指向上一个结点，但这是不可行的，因为每做一次删除都需要对其进行更新。取而代之，我们将链表中的每一个结点都添加一个指向上一项的链接。如图3-4所示，这称为**双向链表**（`doubly linked list`）。

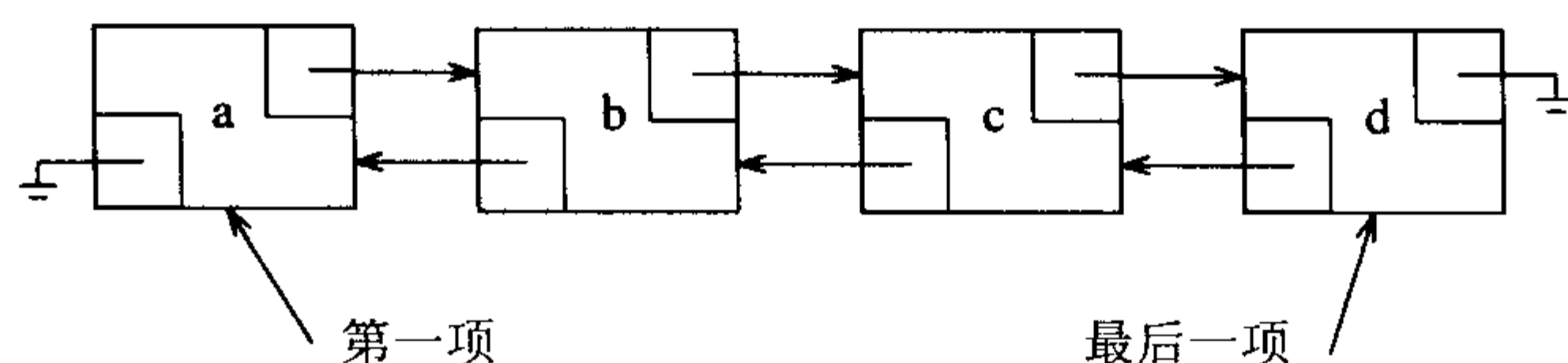


图3-4 双向链表

### 3.3 STL中的向量和表

在C++语言的库中包含有公共数据结构的实现。C++中的这部分内容就是众所周知的**标准模板库**（`Standard Template Library, STL`）。表ADT就是在STL中实现的数据结构之一。其他的数据结构我们将在第4章中介绍。一般来说，这些数据结构称为**集合**（`collection`）或**容器**（`container`）。

表ADT有两个流行的实现。`vector`给出了表ADT的可增长的数组实现。使用 `vector` 的优点在于其在常量的时间里是可索引的。缺点是插入新项或删除已有项的代价是昂贵的，除非是这些操作发生在 `vector` 的末尾。`list` 提供了表ADT的双向链表实现。使用 `list` 的优点是，如果变化

发生的位置已知的话，插入新项和删除已有项的代价是很小的。缺点是list不容易索引。vector和list两者在查找时效率都很低。在本讨论中，list总是指STL中的双向链表，而“表”则是指更一般的ADT表。

vector和list两者都是用其包含的项的类型来例示的类模板。两者都有几个公共的方法。

74 所示的前三个方法事实上对所有的STL容器都适用：

- `int size() const`: 返回容器内的元素个数。
- `void clear()`: 删除容器中所有的元素。
- `bool empty()`: 如果容器没有元素，返回true，否则返回false。

vector和list两者都支持在常量的时间内在表的末尾添加或删除项。vector和list两者都支持在常量的时间内访问表的前端的项。这些操作如下：

- `void push_back( const Object & x )`: 在表的末尾添加x。
- `void pop_back()`: 删除表的末尾的对象。
- `const Object & back() const`: 返回表的末尾的对象（也提供返回引用的修改函数）。
- `const Object & front() const`: 返回表的前端的对象（也提供返回引用的修改函数）。

因为双向链表允许在表的前端进行高效的改变，但是vector不支持，所以，下面的两个方法仅对list有效：

- `void push_front( const Object & x )`: 在list的前端添加x。
- `void pop_front()`: 在list的前端删除对象。

vector也有list所不具有的特有的方法。有两个方法可以进行高效的索引。另外两个方法允许程序员观察和改变vector的内部容量。这些方法是：

- `Object & operator[] (int idx)`: 返回vector中idx索引位置的对象，不包含边界检测（也提供返回常量引用的访问函数）。
- `Object & at ( int idx)`: 返回vector中idx索引位置的对象，包含边界检测（也提供返回常量引用的访问函数）。
- `int capacity() const`: 返回vector的内部容量（详细信息参见3.4节）。
- `void reserve( int new Capacity )`: 设定vector的新容量。如果已有良好的估计的话，这可以避免对vector进行扩展（详细信息见3.4节）。

### 3.3.1 迭代器

一些表的操作，例如那些在表的中部进行插入和删除的操作，需要位置标记。在STL中，通过内置类型iterator来给出位置。特别地，对list<string>，位置通过类型list<string>::iterator给出；对vector<int>，位置由类vector<int>::iterator给出；依此类推。在描述某些方法的时候，为简明起见，我们使用iterator。但是，在编写代码的时候还是使用实际的嵌套类名称。

在开始的时候，需要处理三个问题：第一，如何得到迭代器；第二，迭代器可以执行什么操作；第三，哪些表ADT方法需要迭代器作为形参。

#### 1. 获得迭代器

对第一个问题，STL表（包括其他STL容器）定义了一对方法：

- `iterator begin()`: 返回指向容器的第一项的一个适当的迭代器。
- `iterator end()`: 返回指向容器的终止标志（容器中最后一项的后面的位置）的一个适当的迭代器。

end方法看起来有点不寻常，因为它返回的迭代器指向容器的“边界之外”。为研究这个问题，考察下面典型的用于打印vector v的项的代码：

```
for( int i=0; i!=v.size(); ++i)
    cout<< v[i]<<endl;
```

如果使用迭代器重写代码，那么自然就有对应的使用begin和end方法的代码。

```
for( vector<int>::iterator itr = v.begin();itr != v.end();itr.???)
    cout<<itr.???<<endl;
```

在循环终止检测中，i!=v.size()和itr!=v.end()两者都试图检测循环计数器是否已经“超出边界”。这段代码引出了第二个问题，迭代器必须有关联的方法（这里未知的方法由???代替）。

## 2. 迭代器方法

由上述的代码片段，可以很明显地看到，迭代器可以使用!=和==进行比较，很可能也有复制构造函数和operator=的定义。因此，迭代器有许多方法，并且许多方法都使用操作符重载。除了复制以外，迭代器的最常见的操作如下：

- itr++和++itr：推进迭代器itr至下一个位置。前缀和后缀两种形式都是允许的。
- \*itr：返回存储在迭代器itr指定位置的对象的引用。返回的引用或许能、或许不能被修改（后面我们将讨论这些细节）。
- itr1==itr2：如果itr1和itr2都指向同一个位置就返回true，否则，返回false。
- itr1!=itr2：如果itr1和itr2都指向不同位置就返回true，否则，返回false。

使用这些操作符，打印程序代码将为：

```
for( vector<int>::iterator itr= v.begin(); itr !=v.end(); ++itr)
    cout<<*itr<<endl;
```

使用操作符重载可以允许迭代器访问当前项，然后使用\*itr++推进到下一项。于是，上述代码片段也可以改写如下：

```
vector<int>::iterator itr= v.begin();
while( itr !=v.end())
    cout<<*itr++<<endl;
```

## 3. 需要迭代器的容器操作

对于最后一个问题，需要使用迭代器的三个流行的方法，如下所示，这些方法在表（vector或list表）的特定位置添加或删除项。

- iterator insert( iterator pos, const Object & x)：添加x到表中迭代器pos所指向的位置之前的位置。这对list是常量时间操作，但是对vector则不是。返回值是一个指向插入项位置的迭代器。
- iterator erase( iterator pos)：删除迭代器所给出位置的对象。这对list来说是常量时间操作，但对vector则不是。返回值是调用之前pos所指向元素的下一个元素的位置。这个操作使pos失效。pos不再有用，因为它所指向的容器变量已经被删除了。
- iterator erase( iterator start, iterator end)：删除所有的从位置start开始、直到位置end（但是不包括end）的所有元素。注意，整个表的删除可以调用c.erase( c.begin(), c.end())。

### 3.3.2 示例：对表使用erase

作为一个示例，这里给出一个例程，从表的起始项开始间隔地删除项。这样，如果表包含6、



5、1、4和2，那么，调用该方法后，表将只包含5和4。这是通过遍历表，并对每个第二项使用erase方法实现的。对于list，这是一个线性时间例程，因为每次调用erase都消耗常量的时间。但是对于vector整个例程将消耗平方级的时间，因为每次调用erase都是低效的，花费 $O(N)$ 时间。因此，正常情况下我们只为list编写代码。为实验的目的，我们编写一个可以同时用于list和vector的通用函数模板，并且提供消耗时间的信息。图3-5所示是该函数模板。之所以在第4行使用typename，是因为一些编译器需要声明Container::iterator是一个类型而不是一个数据字段或者方法。如果运行代码并传递list<int>，那么，对400 000项的list，程序将花费0.062s的时间；对800 000项的list，程序将花费0.125s的时间。很明显，这是线性时间例程，因为程序的运行时间与数据的输入量按相同的速率增长。当传递vector<int>时，对400 000项的vector将花费差不多两分半钟的时间；对800 000项的vector，例程将花费超过十分钟的时间。当输入量以两倍的速率增长时，运行时间以四倍的速率增长。这是二次算法的表征。

```

1  template <typename Container>
2  void removeEveryOtherItem( Container & lst )
3  {
4      typename Container::iterator itr = lst.begin( );
5      while( itr != lst.end( ) )
6      {
7          itr = lst.erase( itr );
8          if( itr != lst.end( ) )
9              ++itr;
10     }
11 }
```

图3-5 使用迭代器来间隔地删除表中的项(vector或list)。对list是高效的，对vector则不是

### 3.3.3 const\_iterator

\*itr的结果不只是迭代器指向的项的值，也是该项本身。这个区别使得迭代器的功能很强，但也使其更复杂。为研究其优点，假设我们需要将一个集合里的所有项都改为一个特殊的值。下面的例程工作于vector和list，并且按线性时间运行。这是编写类型无关的泛型代码的一个极好的例子。

77

```

template<typename Container, typename Object>
void change( Container & c, const Object & newValue)
{
    typename Container::iterator itr=c.begin();
    while( itr != c.end())
        *itr++ = newValue;
}
```

为研究潜在的问题，假设Container c通过常量引用传递至例程。这意味着我们不期望对c有任何的改变，同时编译器要通过禁止调用c的任何修改函数来确保这一点。考察下面打印的整数list的代码。该代码试图暗中修改list：

```

void print( const list<int> & lst, ostream & out = cout)
{
    typename Container::iterator itr = lst.begin();
    while( itr != lst.end())
    {
        out << *itr <<endl;
        *itr=0;    // This is fishy !!!
        itr++;
    }
}
```

```

    }
}

```

如果这段代码是合法的话，那么list的定常性就完全没有任何意义了，因为改变起来太容易了。这段代码是非法的，并且不会被编译。STL提供的解决方案是每一个集合不仅包含嵌套的iterator类型，也包含嵌套的const\_iterator类型。iterator和const\_iterator之间的主要区别是：const\_iterator的operator\*返回常量引用，这样const\_iterator的\*itr就不能出现在赋值语句的左边。

78

更进一步地，编译器还会要求必须使用const\_iterator类遍历常量集合。如下所示，有两个版本的begin和两个版本的end来实现这个功能：

- iterator begin()
- const\_iterator begin() const
- iterator end()
- const\_iterator end() const

两个版本的begin可以在同一个类里面，因为方法（例如，无论是访问函数还是修改函数）的定常性被认为是标号的一部分。在1.7.2节我们曾遇到过这个窍门，在3.4节还将看到这个窍门。两者都是出现在重载operator[]的情况下。

如果对非常量容器调用begin，那么返回iterator的“修改函数”版本就将被调用。然而，如果对返回const\_iterator的常量容器调用begin，那么返回值就可能没有被赋值为iterator。如果尝试这么做的话，编译错误就会产生。一旦itr是const\_iterator，\*itr=0很容易就会被确定是非法的。

图3-6的代码为使用const\_iterator打印任意集合的例子。集合中的项打印在括号中，并用逗号隔开。

```

1  template <typename Container>
2  void printCollection( const Container & c, ostream & out = cout )
3  {
4      if( c.empty( ) )
5          out << "(empty)";
6      else
7      {
8          typename Container::const_iterator itr = c.begin( );
9          out << "[" << *itr++; // Print first item
10
11         while( itr != c.end( ) )
12             out << ", " << *itr++;
13         out << "]" << endl;
14     }
15 }

```

图3-6 打印任何容器

## 3.4 向量的实现

在本节中，给出了一个可用的vector类模板的实现。vector是基本类类型，这意味着不同于C++中的基本数组，vector可以复制并且其占用的内存可以自动回收（通过其析构函数）。在1.5.6节，我们已经讨论了C++基本数组的一些重要特性：

- 数组就是指向一块内存的指针变量；实际的数组的大小必须由程序员单独确定。

79

- 内存块可以通过new[]来分配，但是相应地也就必须用delete[]来释放。
- 内存块的大小不能改变（但是可以定义一个新的具有更大内存块的数组，并且用原来的数组来将其初始化，然后原来的内存块就可以释放了）。

为避免与库函数类相混淆，我们的类模板命名为Vector。在研究（少于一百行的）Vector代码前，先概括其主要的细节。

(1) Vector将仍然是基本数组（通过一个指针变量来指向分配的内存块）。数组的容量和当前的数组项数目存储在Vector里。

(2) Vector将通过实现“三大函数”，为复制构造函数和operator=提供深复制，同时也提供析构函数来回收基本数组。

(3) Vector将提供resize例程来改变Vector的大小（通常是更大的数）；提供reserve例程来改变Vector的容量（通常是更大的数）。容量的改变是通过为基本数组分配一个新的内存块，然后复制旧内存块的内容到新块中，再释放旧块的内存来实现的。

(4) Vector将提供operator[]的实现（正如1.7.2节中所提到的，operator[]典型的实现有访问函数和修改函数两个版本）。

(5) Vector将提供基本的例程，例如size、empty、clear（它们是典型的一行例程）、back、pop\_back和push\_back。如果大小和容量都是一样的话，push\_back例程将调用reserve来增大Vector的容量。

(6) Vector将支持嵌套的iterator和const\_iterator类型，并且提供关联的begin和end方法。

图3-7和图3-8显示了Vector类。作为STL的副本，Vector类也有有限的错误检验。稍后我们将主要讨论如何提供错误检验的功能。

正如在第90~92行所显示的，Vector将其作为数据成员来存储大小、容量和基本数组。在第5~7行的构造函数允许使用者自己定义初始大小(默认值为0)。然后初始化数据成员，并令容量比大小稍大一点。这样就可以在不改变容量的前提下执行push\_backs。

第8~9行显示的复制构造函数调用operator=对已有的Vector进行复制。第10~11行的析构函数回收基本数组的内存空间。最巧妙的例程是第13~26行显示的operator=。在第15行的混淆检验之后，在第17行释放旧数组，在第21行生成与所复制的Vector同样容量的新数组。在大小被复制后，依次复制数据项。当然，代码时常不必要地进行释放和分配新的数组的操作，因为原始的数组有可能是足够大的。因为内存的重新分配的代价是很显著的，所以，对库Vector类编写额外的逻辑是值得的。

第28~33行是resize例程。在对容量进行扩展后，代码简单地设定数据成员theSize。扩展容量的代价是高昂的，因此，如果容量进行扩展，除非大小也显著增长（大小为0时，使用+1），否则就将容量扩展为大小的两倍，以避免对容量进行再次扩展。如第35~49行所示，容量的扩展是通过reserve例程来实现的。该例程具有与operator[]大部分相同的逻辑：在第42行分配一个新数组，在第43和44行复制旧数组的内容，然后在第48行回收旧数组。如第37和38行所示，reserve例程可以用来缩小基本数组。但是，所指定的新的容量必须至少和大小一样大，否则，reserve的请求就被忽略。

如第50~53行所示，两个版本的operator[]很简单（事实上，与1.7.2节中的matrix类中operator[]实现非常相似）。通过确定index是否在0至size()-1的范围内（包括size()-1），错误检测功能可以很容易地实现。如果没在这个范围内就抛出一个异常。



```

1  template <typename Object>
2  class Vector
3  {
4  public:
5      explicit Vector( int initSize = 0 )
6          : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
7          { objects = new Object[ theCapacity ]; }
8      Vector( const Vector & rhs ) : objects( NULL )
9          { operator=( rhs ); }
10     ~Vector( )
11         { delete [ ] objects; }
12
13     const Vector & operator= ( const Vector & rhs )
14     {
15         if( this != &rhs )
16         {
17             delete [ ] objects;
18             theSize = rhs.size( );
19             theCapacity = rhs.theCapacity;
20
21             objects = new Object[ capacity( ) ];
22             for( int k = 0; k < size( ); k++ )
23                 objects[ k ] = rhs.objects[ k ];
24         }
25         return *this;
26     }
27
28     void resize( int newSize )
29     {
30         if( newSize > theCapacity )
31             reserve( newSize * 2 + 1 );
32         theSize = newSize;
33     }
34
35     void reserve( int newCapacity )
36     {
37         if( newCapacity < theSize )
38             return;
39
40         Object *oldArray = objects;
41
42         objects = new Object[ newCapacity ];
43         for( int k = 0; k < theSize; k++ )
44             objects[ k ] = oldArray[ k ];
45
46         theCapacity = newCapacity;
47
48         delete [ ] oldArray;
49     }

```

图3-7 Vector类（两部分中的第一部分）

第55~73行是许多小例程的实现，包括：empty、size、capacity、push\_back、pop\_back和back。在第66行是后缀操作符++，该操作符使用theSize来索引数组，然后自增theSize。在讨论迭代器的时候我们见到过相同的习惯用法：\*itr++使用itr来确定访问哪一项，然后推进itr。++的定位：前缀操作符++的情况，\*++itr首先推进itr，然后用新的itr来确定访问哪一项；相似地，object[++theSize]将自增theSize并且使用这个新值来索引数组（这不是我们想要的）。pop\_back和back都会受益于错误检测，因为检测的时候如果大小为零就会抛出一个异常。

```

50  Object & operator[] ( int index )
51      { return objects[ index ]; }
52  const Object & operator[] ( int index ) const
53      { return objects[ index ]; }
54
55  bool empty( ) const
56      { return size( ) == 0; }
57  int size( ) const
58      { return theSize; }
59  int capacity( ) const
60      { return theCapacity; }
61
62  void push_back( const Object & x )
63      {
64          if( theSize == theCapacity )
65              reserve( 2 * theCapacity + 1 );
66          objects[ theSize++ ] = x;
67      }
68
69  void pop_back( )
70      { theSize--; }
71
72  const Object & back ( ) const
73      { return objects[ theSize - 1 ]; }
74
75  typedef Object * iterator;
76  typedef const Object * const_iterator;
77
78  iterator begin( )
79      { return &objects[ 0 ]; }
80  const_iterator begin( ) const
81      { return &objects[ 0 ]; }
82  iterator end( )
83      { return &objects[ size( ) ]; }
84  const_iterator end( ) const
85      { return &objects[ size( ) ]; }
86
87  enum { SPARE_CAPACITY = 16 };
88
89  private:
90      int theSize;
91      int theCapacity;
92      Object * objects;
93  };

```

图3-8 Vector类（两部分中的第二部分）

最后，在第75~85行是内置类型的iterator和const\_iterator的声明以及两个begin方法和两个end方法的声明。这段代码利用了C++中指针变量具有我们所期望的iterator应该具有的所有操作符的事实。指针变量可以复制和比较。\*操作符取得指针变量所指向的对象。最特别地，当++应用到指针变量的时候，如果指针是指向数组内部的，指针变量就会指向下一个顺序存储的对象——增加指针的值使之指向下一个数组元素。这些C++所遵循的指针的语法可以追逆到20世纪70年代早期的C编程语言，C++就是在C语言的基础上发展起来的。STL迭代器的机制是模拟指针的操作而设计。

因此，在第75~76行，声明iterator和const\_iterator的typedef语句很简单地就是指针变量的别名，并且begin和end需要简单地分别返回代表第一个数组位置的内存地址和第一个无

效的数组位置。

vector类型的迭代器和指针的相似性意味着使用vector而不使用C++数组会导致稍高一点的资源消耗。正如所提到的，其缺点是代码没有错误检测。如果迭代器itr冲出了末端标记的话，++itr或\*itr都不需要标识错误。解决这个问题需要使iterator和const\_iterator是真正的嵌套类类型而不是简单的指针变量。使用嵌套类类型是很普通的，这也是在3.5节List类的部分所要讨论的。

## 3.5 表的实现

在本节中，提供了一个可用的list类模板的实现。和vector类中的情况一样，我们的表类还是命名为List以避免与库类的混淆。

回顾一下，前面提到的List类将要作为双向链表来实现，并且我们需要修改指向表两端的指针。只要操作是发生在已知位置，这样做就可以保证每个操作的时间消耗为常量。这个已知位置可以是末尾，也可以是迭代器指定的位置。

考虑到设计需要，我们需要提供下面的4个类：

(1) List类本身。包含连接到表两端的链接、表的大小以及一系列的方法。

(2) Node类。该类看起来像是私有的嵌套类。一个结点包含数据和用来指向其前和其后的结点的指针，以及适当的构造函数。

(3) const\_iterator类。该类抽象了位置的概念，是一个公有的嵌套类。const\_iterator存储指向当前结点的指针，并且提供基本迭代器操作的实现，以及所有的重载操作符，例如=、==、!=和++。

(4) iterator类。该类抽象了位置的概念，是一个公有的嵌套类。除了operator\*操作返回所指向项的引用，而不是该项的常量引用的功能外，iterator具有与const\_iterator相同的功能。一个重要的技术点是iterator可以用于任何需要使用const\_iterator的例程里，反之则不是。换句话说，iterator就是const\_iterator。

因为迭代器类存储指向“当前结点”的指针，并且尾部标志是一个有效的位置，这使得在表的末尾添加一个额外的结点来作为尾部标志成为可能。进一步地，也可以在表的前端生成一个额外的结点，从而逻辑上作为开始标志。这些额外的结点有时被称为哨兵结点；特别地，在头部的结点有时候称为表头结点（header node），而在末端的结点称为尾结点（tail node）。

使用这些额外结点的好处是可以去掉很多特例，这极大地简化了程序代码。例如，如果我们不使用表头结点，那么删除第一个结点将成为一个特例，因为我们必须在删除过程中重新设置表的链接到第一个结点上，而且删除算法一般来说也需要访问被删除结点前面的结点（没有表头结点，第一个结点前面就没有结点了）。图3-9是带有表头结点和尾结点的双向链表。图3-10是空双向链表。

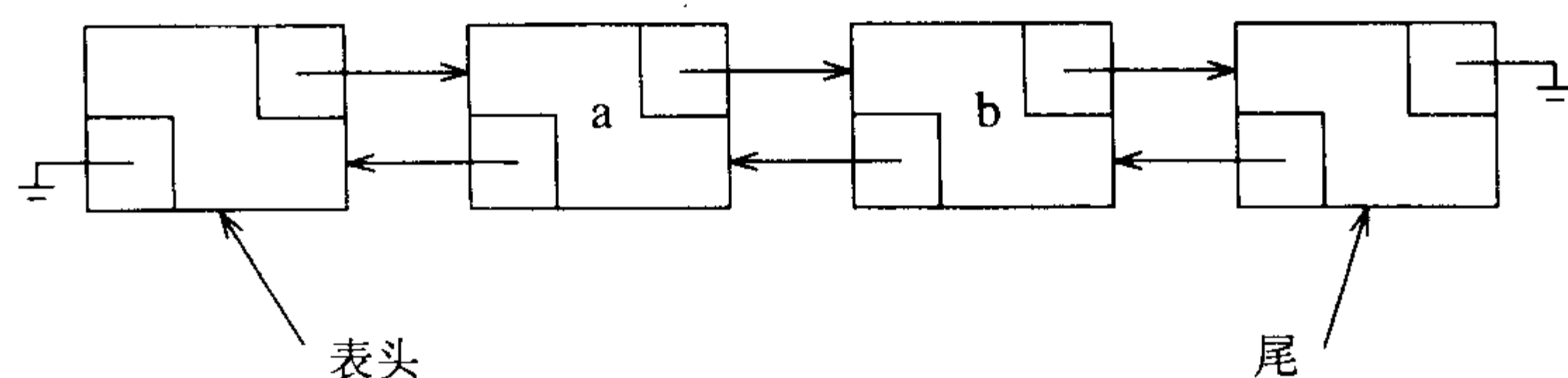


图3-9 具有表头结点和尾结点的双向链表



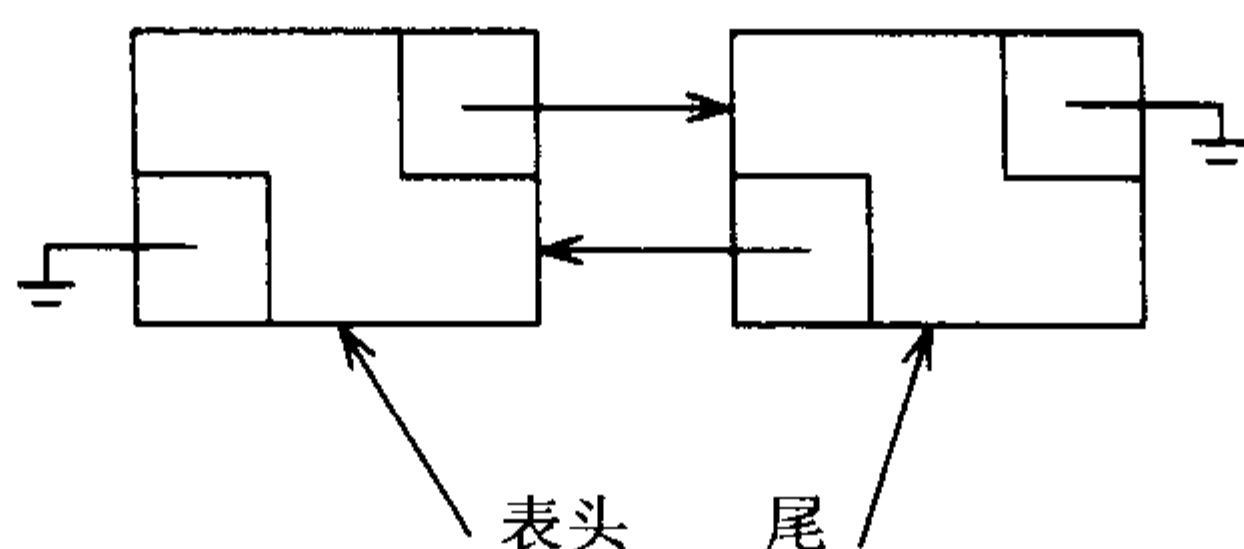


图3-10 具有表头结点和尾结点的空双向链表

84

图3-11和图3-12是List类的概要和部分实现。

```

1  template <typename Object>
2  class List
3  {
4      private:
5          struct Node
6              { /* See Figure 3.13 */ };
7
8      public:
9          class const_iterator
10             ( /* See Figure 3.14 */ );
11
12         class iterator : public const_iterator
13             { /* See Figure 3.15 */ };
14
15     public:
16         List()
17             { /* See Figure 3.16 */ }
18         List( const List & rhs )
19             { /* See Figure 3.16 */ }
20         ~List( )
21             { /* See Figure 3.16 */ }
22         const List & operator= ( const List & rhs )
23             { /* See Figure 3.16 */ }
24
25         iterator begin( )
26             { return iterator( head->next ); }
27         const_iterator begin( ) const
28             { return const_iterator( head->next ); }
29         iterator end( )
30             { return iterator( tail ); }
31         const_iterator end( ) const
32             { return const_iterator( tail ); }
33
34         int size( ) const
35             { return theSize; }
36         bool empty( ) const
37             { return size( ) == 0; }
38
39         void clear( )
40         {
41             while( !empty( ) )
42                 pop_front ( );
43     }

```

图3-11 List类（两部分中的第一部分）

观察第5行私有嵌套的Node类的声明的开始部分，可以看到这里使用了struct而不是使用class关键字。在C++中，struct是C编程语言遗留下来的产物。C++中的struct在本质上来说

就是其成员默认为公有的class。回顾一下，在class中，其成员都是私有的。很明显，struct关键字不是必需的，然而，还是经常可以看到程序员在声明一个大部分数据需要直接访问而不是使用方法来访问的类型时使用struct。在我们的例子中，令Node类中的成员为public并不会成为什么问题，因为Node类本身是私有的并且不允许来自List类之外的访问。

```

44     Object & front( )
45     { return *begin( ); }
46     const Object & front( ) const
47     { return *begin( ); }
48     Object & back( )
49     { return *--end( ); }
50     const Object & back( ) const
51     { return *--end( ); }
52     void push_front( const Object & x )
53     { insert( begin( ), x ); }
54     void push_back( const Object & x )
55     { insert( end( ), x ); }
56     void pop_front( )
57     { erase( begin( ) ); }
58     void pop_back( )
59     { erase( --end( ) ); }
60
61     iterator insert( iterator itr, const Object & x )
62     { /* See Figure 3.18 */ }
63
64     iterator erase( iterator itr )
65     { /* See Figure 3.20 */ }
66     iterator erase( iterator start, iterator end )
67     { /* See Figure 3.20 */ }
68
69     private:
70         int    theSize;
71         Node *head;
72         Node *tail;
73
74         void init( )
75         { /* See Figure 3.16 */ }
76 };

```

图3-12 List类（两部分中的第二部分）

观察第9行的公有嵌套的const\_iterator类的声明的开始部分和第12行公有嵌套的iterator类的声明的开始部分，可以看到这里有一个不常见的语法inheritance（这是一个功能强大的结构，在本书的其他部分则没有用到）。继承语法是说，iterator具有与const\_iterator完全相同（有可能会多一些）的功能，并且iterator与const\_iterator的数据类型是完全兼容的。所有需要使用const\_iterator的地方都可使用iterator。在以后遇到实际的实现的时候我们再讨论这些细节。

第70~72行是List的数据成员，命名了指向表头结点和尾结点的指针。我们也将记录数据成员的大小，这样一来size方法就可以在常量的时间内实现。

List类的其他部分包括构造函数、三大函数和一些方法。许多的方法都是一行的。begin和end返回适当的迭代器；第26行的调用是一个典型的实现，在实现中返回一个已构造的迭代器（这样iterator和const\_iterator类每一个都有自己的构造函数，该构造函数使用指向Node的指针作为参数）。

第39~43行的clear方法执行的时候重复地执行删除操作来删除每一项，直到List变为空的

为止。使用这种策略可以避免clear染指结点空间的回收。现在结点空间的回收已经归入pop\_front来执行。第44~59行的方法都是通过巧妙地包含和使用恰当的迭代器来工作的。回顾一下，insert方法在某个位置之前插入，因此，在需要的时候，push\_back在末尾标记之前插入。对于pop\_back，注意erase(--end())生成了一个对应末尾标记的临时迭代器，后移这个临时迭代器，然后使用这个迭代器来执行erase。back也使用相似的工作原理。注意，在pop\_front和pop\_back操作时，我们再一次避免使用结点回收。

图3-13是Node类。该类包括所存储的项、指向Node之前及之后的指针和一个构造函数。所有的数据成员都是公有的。

```

1 struct Node
2 {
3     Object data;
4     Node *prev;
5     Node *next;
6
7     Node( const Object & d = Object( ), Node *p = NULL, Node *n = NULL )
8         : data( d ), prev( p ), next( n ) { }
9 };

```

图3-13 List类的嵌套的Node类

85 图3-14是const\_iterator类；图3-15是iterator类。正如我们早前提到过的，图3-15中第39行的语法即是称为继承性的高级特性。这也就意味着iterator就是const\_iterator。当iterator类使用这样的方式来编写时，该类就从const\_iterator继承了所有的数据和方法。然后就可以对iterator类添加新的数据或添加新的方法，以及覆盖（例如重新定义）已有的方法。在最一般的情况下，将会产生显著的语法包袱（常常致使关键字virtual出现在代码中）。

然而，在本例中，这许多的语法包袱是可以避免的，因为我们既不添加新的数据，又不试图改变已有方法的运作。但是，我们添加一些新的方法到iterator类中（用与const\_iterator类中已有的方法非常相似的符号）。这样一来就避免了使用virtual。虽然如此，在const\_iterator中还是有相当数量的语法技巧。

86 在第28和29行，const\_iterator像存储它的单一数据成员一样存储指向“当前”结点的指针。  
87 一般地，这个指针都是私有的，但是如果是私有的，那么iterator将不能访问这个指针。令const\_iterator的成员为protected的，将允许从const\_iterator继承的类具有访问这些成员的权限，但是不允许其他的类访问。

在第34和35行是const\_iterator的构造函数。该构造函数在List类的begin和end的实现中用到。我们不希望所有的类都能访问这个构造函数（假定迭代器不能从指针变量显式构造），因此该构造函数不可以是公有的，但是我们又希望iterator类可以访问它，因此，逻辑上说，这个构造函数是被保护的。然而，这个保护没有提供List类访问这个构造函数的权限。解决的方案是第37行的友元声明（friend declaration）。该声明允许List类访问const\_iterator的非公有成员。

const\_iterator的公有方法都使用操作符重载。operator==、operator!=和operator\*是最直接的。在第10~21行，可以看到operator++的实现。回想在语法上前缀和后缀版本的operator++是完全不同的。因此，需要对不同的形式分别来编写例程。它们拥有相同的名字，因此必须用不同的符号来区分。C++需要通过给前缀形式指定空参数表，给后缀形式指定一个（匿名的）int参数来赋予前缀和后缀形式以不同的标识。然后，++itr调用零参数operator++；而



itr++调用单参数operator++。这个int参数永远也不使用，其存在的意义仅仅在于给出一个不同的标识。实现指出在许多可以选择使用前缀或使用后缀operator++的情况下，使用前缀形式要快于使用后缀形式。

```

1 class const_iterator
2 {
3     public:
4         const_iterator( ) : current( NULL )
5             { }
6
7         const Object & operator* ( ) const
8             { return retrieve( ); }
9
10        const_iterator & operator++ ( )
11        {
12            current = current->next;
13            return *this;
14        }
15
16        const_iterator operator++ ( int )
17        {
18            const_iterator old = *this;
19            ++( *this );
20            return old;
21        }
22
23        bool operator== ( const const_iterator & rhs ) const
24            { return current == rhs.current; }
25        bool operator!= ( const const_iterator & rhs ) const
26            { return !( *this == rhs ); }
27
28        protected:
29            Node *current;
30
31            Object & retrieve( ) const
32                { return current->data; }
33
34            const_iterator( Node *p ) : current( p )
35                { }
36
37            friend class List<Object>;
38 };

```

图3-14 List类的嵌套的const\_iterator类

在iterator类中，受保护的构造函数（在第64行）使用一个初始化表来初始化继承来的当前的结点。我们的确不需要重新实现operator==和operator!=，因为它们都是未经任何改变就继承下来的。我们提供了一对新的operator++的实现（因为改变了返回类型），该新的实现隐藏了const\_iterator的原始实现，同时也提供了operator\*的一个访问函数/修改函数对。第47~48行显示的访问函数operator\*简单地使用了与const\_iterator完全相同的实现。在iterator中，修改函数必须显式实现，因为不这样的话原始的实现就会被新加的修改函数隐藏。

图3-16是构造函数和三大函数。因为零参数构造函数和复制构造函数两者都必须分配表头结点和尾结点，我们给出了一个私有的init例程。init生成了一个空List。析构函数回收表头结点和尾结点；所有的其他结点在析构函数调用clear时回收。相似地，operator=是通过调用公有方法来实现的，而没有试图使用低级的指针操作。

```

39 class iterator : public const_iterator
40 {
41     public:
42         iterator( )
43         { }
44
45         Object & operator* ( )
46         { return retrieve( ); }
47         const Object & operator* ( ) const
48         { return const_iterator::operator*( ); }
49
50         iterator & operator++ ( )
51         {
52             current = current->next;
53             return *this;
54         }
55
56         iterator operator++ ( int )
57         {
58             iterator old = *this;
59             ++( *this );
60             return old;
61         }
62
63     protected:
64         iterator( Node *p ) : const_iterator( p )
65         { }
66
67     friend class List<Object>;
68 };

```

图3-15 List类的嵌套的iterator类

```

1 List( )
2 { init( ); }
3
4 ~List( )
5 {
6     clear( );
7     delete head;
8     delete tail;
9 }
10
11 List( const List & rhs )
12 {
13     init( );
14     *this = rhs;
15 }
16
17 const List & operator= ( const List & rhs )
18 {
19     if( this == &rhs )
20         return *this;
21     clear( );
22     for( const_iterator itr = rhs.begin( ); itr != rhs.end( ); ++itr )
23         push_back( *itr );
24     return *this;
25 }
26

```

图3-16 List类的构造函数、三大函数和私有init例程

```

27 void init( )
28 {
29     theSize = 0;
30     head = new Node;
31     tail = new Node;
32     head->next = tail;
33     tail->prev = head;
34 }

```

图3-16 List类的构造函数、三大函数和私有init例程（续）

图3-17例举了一个包含x的新结点是如何与通过p和p.prev指向的结点结合的。结点指针的赋值可以按下面的方式编写：

```

Node *newNode = new Node( x, p->prev, p); // Steps 1 and 2
p->prev->next = newNode;                 // Step 3
p->prev = newNode;                       // Step 4

```

第3步和第4步可以合并，于是仅得到两行：

```

Node *newNode = new Node( x, p->prev, p); // Steps 1 and 2
p->prev = p->prev->next = newNode;        // Steps 3 and 4

```

但是这两行还是可以合并，得到：

```
p->prev = p->prev->next = new Node( x, p->prev, p);
```

这使得图3-18的insert例程更为简短。

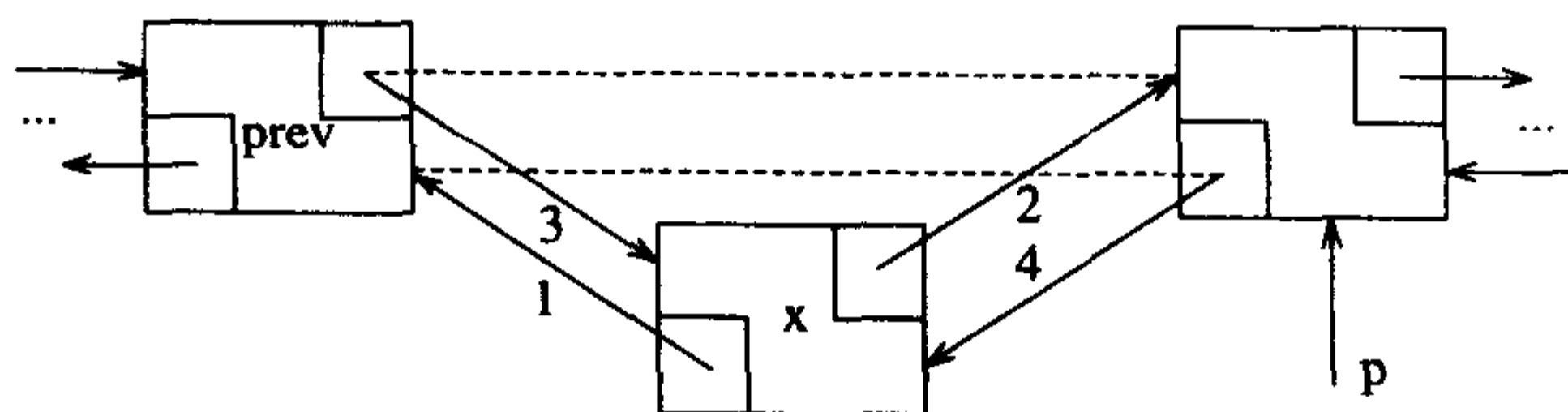


图3-17 将新结点插入双向链表然后按指示的顺序改变指针

```

1 // Insert x before itr.
2 iterator insert( iterator itr, const Object & x )
3 {
4     Node *p = itr.current;
5     theSize++;
6     return iterator( p->prev = p->prev->next = new Node( x, p->prev, p ) );
7 }

```

图3-18 List类的insert例程

图3-19是删除结点的逻辑。如果p指向要被删除的结点，那么在该结点被回收前，只有两个指针需要改变：

```

p->prev->next = p->next;
p->next->prev = p->prev;
delete p;

```

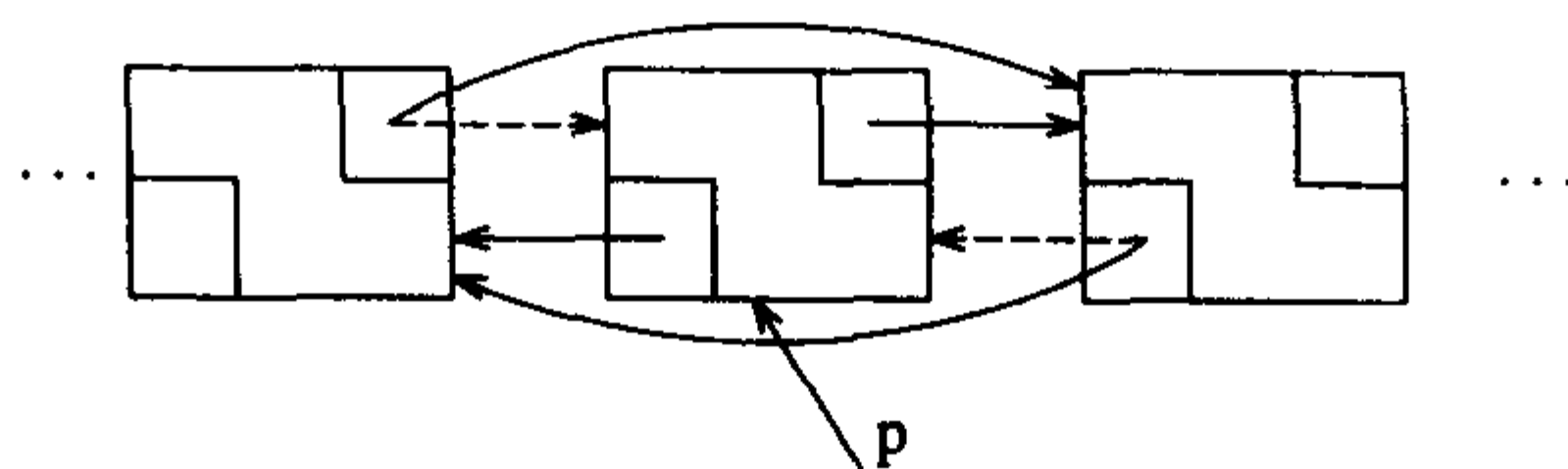


图3-19 从双向链表中删除由p指定的结点



图3-20给出的是一对erase例程。第一个版本的erase例程包含上述的三行代码，并且返回一个指向删除元素后面的项的iterator。和insert一样，erase必须更新theSize。第二个版本的erase仅仅简单使用iterator来调用第一个版本的erase。注意，在第16行的for循环中不可以简单地使用itr++，也不能忽略第17行中erase的返回值。itr的值在调用erase后立刻就失效了，这也是erase返回一个iterator的原因。

```

1 // Erase item at itr.
2 iterator erase( iterator itr )
3 {
4     Node *p = itr.current;
5     iterator retVal( p->next );
6     p->prev->next = p->next;
7     p->next->prev = p->prev;
8     delete p;
9     theSize--;
10
11     return retVal;
12 }
13
14 iterator erase( iterator start, iterator end )
15 {
16     for( iterator itr = from; itr != to; )
17         itr = erase( itr );
18
19     return to;
20 }

```

图3-20 List类的erase例程

92 在分析代码的过程中，我们注意到可能会发生的许多错误，而且没有针对这些可能的错误的检测。例如，传递给erase和insert的迭代器可能没有初始化或者这个迭代器是错误的表的。当迭代器已经指向末尾标记或没有初始化的时候，还可以执行++和\*的操作。

一个没有初始化的迭代器将指向NULL，因此，这种情况是很容易检测的。末尾标记的next指针指向NULL，因此，为进行++和\*操作，检测末尾标记的情况也容易做到。但是为了确定传递给erase和insert的迭代器是否属于正确的表，迭代器就必须存储一个附加的数据成员来提供一个指针用以指向List。整个附加的成员函数就是通过List构造的。

我们将仅勾勒基本的思想，而将具体的实现留做练习。在const\_iterator类，添加一个指向List的指针，并且修改受保护的构造函数，使其以List作为一个参数。我们也添加方法，使得当某个预想的情况没达到时就抛出一个异常。修订的受保护的代码看起来有点像图3-21的

```

1 protected:
2     const List<Object> *theList;
3     Node *current;
4
5     const_iterator( const List<Object> & lst, Node *p )
6         : theList( &lst ), current( p )
7     {
8     }
9
10    void assertIsValid( ) const
11    {
12        if( theList == NULL || current == NULL || current == theList->head )
13            throw IteratorOutOfBoundsException( );
14    }

```

图3-21 具有执行附加错误检验能力的const\_iterator中修订后的受保护部分

代码。现在，所有的对iterator和const\_iterator构造函数的调用将由先前的使用一个参数变为现在的使用两个参数。正如在List中的begin方法一样：

```
const_iterator begin() const
{
    const_iterator itr( *this, head );
    return ++itr;
}
```

然后，insert也可以被修订为看起来有些像如图3-22所示那样的代码。我们将这些修订的细节留做练习。

93

```
1 // Insert x before itr.
2 iterator insert( iterator itr, const Object & x )
3 {
4     itr.assertIsValid( );
5     if( itr.theList != this )
6         throw IteratorMismatchException( );
7
8     Node *p = itr.current;
9     theSize++;
10    return iterator( *this,
11                    p->prev = p->prev->next = new Node( x, p->prev, p ) );
12 }
```

图3-22 带有附加错误检验的List insert

## 3.6 栈ADT

### 3.6.1 栈模型

栈（stack）是限制插入和删除操作只能在一个位置上进行的表，该位置是表的末端，称为栈的顶（top）。对栈的基本操作是push（进栈）和pop（出栈），前者相当于插入，后者则是删除最后插入的元素。最后插入的元素可以通过使用top例程在执行pop之前进行访问。对空栈进行的pop或top操作，在栈ADT中一般认为是一个错误。另一方面，当运行push时在空间之外操作是实现限制，但不是ADT错误。

94

栈有时又称为LIFO（后进先出）表。在图3-23中描述的模型意味着只有push是输入操作，并且只有pop和top是输出操作。普通的清空栈的操作和判断是否空栈的测试都是栈的操作指令系统的一部分，但是，对栈所能够做的所有操作基本上就是push和pop操作。

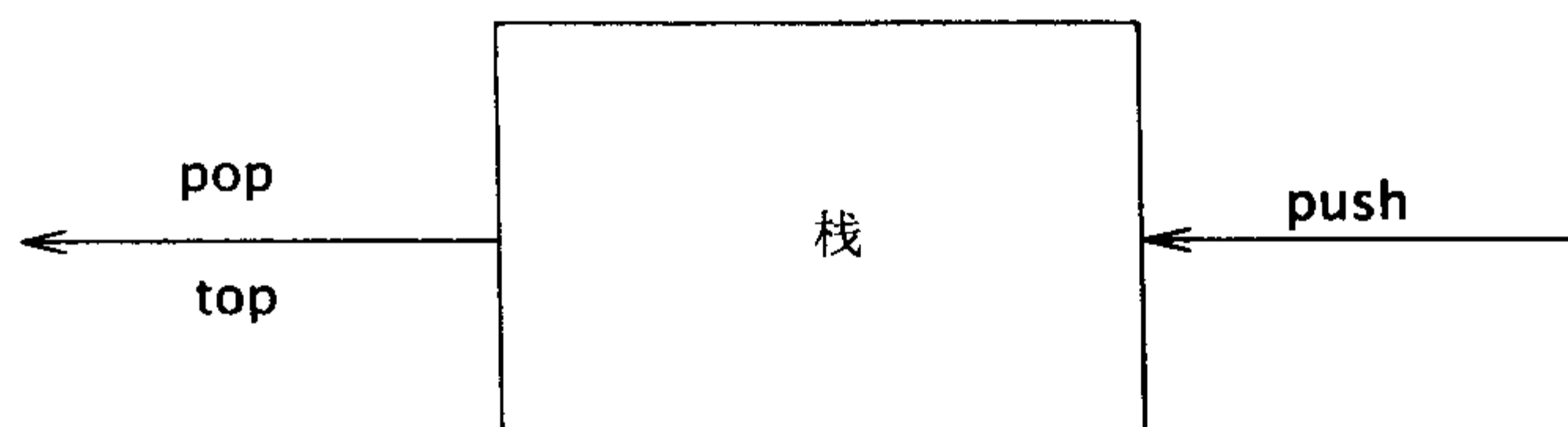


图3-23 栈模型：通过push向栈输入，通过pop和top从栈输出

图3-24所示是进行若干操作后的一个抽象的栈。一般的模型是，存在某个元素位于栈顶，而该元素是唯一的可见元素。



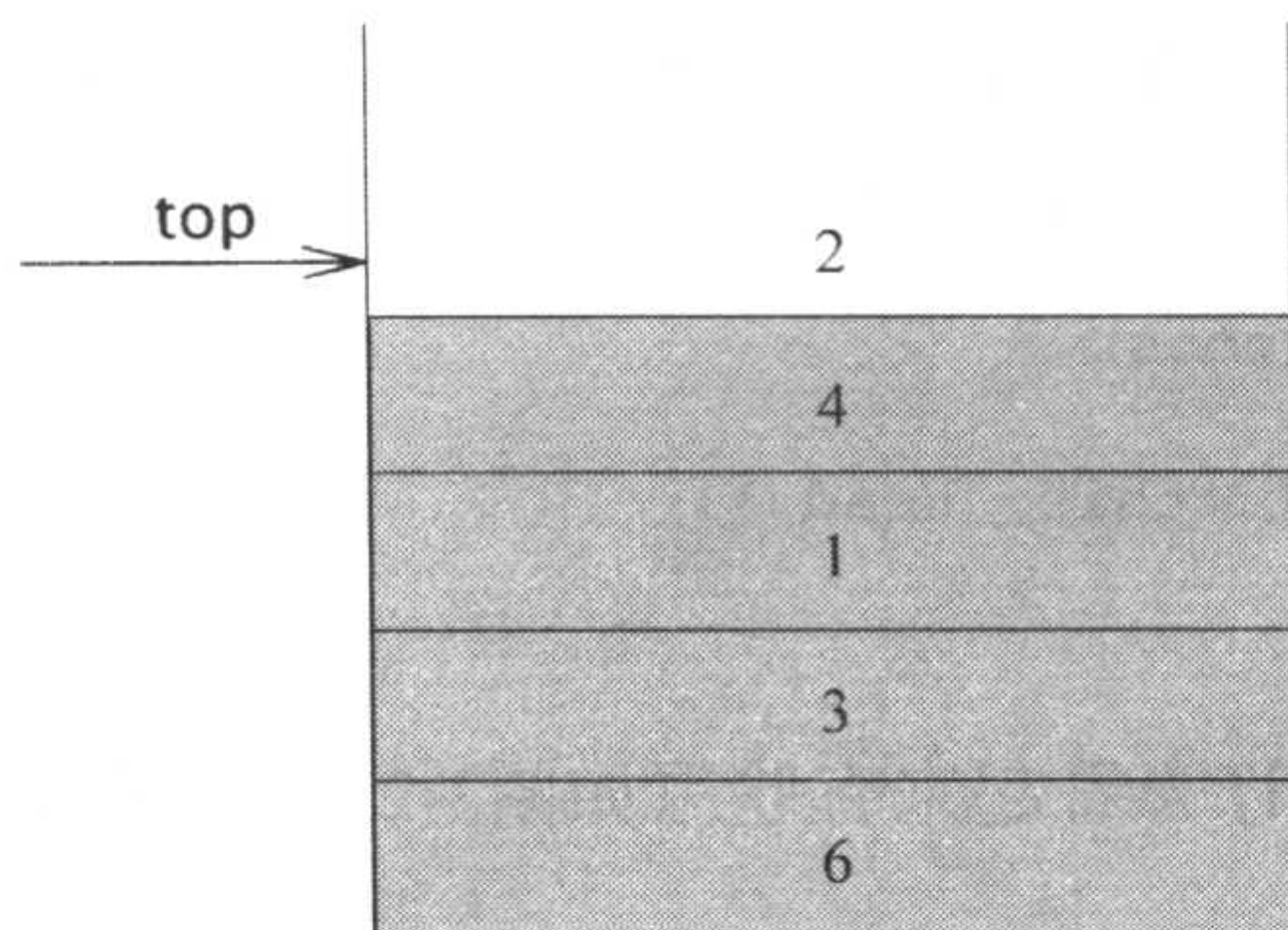


图3-24 栈模型：只有栈顶元素是可访问的

### 3.6.2 栈的实现

由于栈是一个表，因此任何实现表的方法都能实现栈。很明显list和vector支持栈操作；99%的情况下，它们都是最合理的选择。有时候为特殊目的而设计的实现可以运行得更快。由于栈的所有操作都是常量时间的操作，所以，除非是在很特别的环境下，否则不太可能会有明显的改进。

95

针对这些特别的情况，我们将给出两个流行的栈的实现。一个使用链接结构，另一个则使用数组。这两者在vector和list里的逻辑都很简单，所以，这里不提供源代码。

#### 1. 栈的链表实现

栈的第一种实现方法是使用单向链表。我们通过在表顶端插入元素来实现push，通过删除表顶端元素实现pop。top操作只是访问表顶端元素并返回它的值。有时pop操作和top操作合二为一。

#### 2. 栈的数组实现

另一种可选的实现避免了使用链并且可能是更流行的解决方案。由于使用vector中的back、push\_back和pop\_back实现，因此这个实现很简单。每个栈有一个theArray和一个topOfStack，对于空栈其值为-1（这也就是空栈的初始化）。为了将某个元素x压入到栈中，将topOfStack加1，然后置theArray[topOfStack]=x。为了弹出栈元素，置pop函数的返回值为theArray[topOfStack]，然后将topOfStack减1。

注意，这些操作不仅以常数时间运行，而且是以非常快的常数时间运行。在某些机器上，若在带有自增和自减寻址功能的寄存器上操作，则（整数的）push和pop都可以写成一条机器指令。最现代化的计算机将栈操作作为其指令系统的一部分，这个事实强化了这样一种思想，即在计算机科学中，栈很可能是继数组之后的最基本的数据结构。

### 3.6.3 应用

很自然的，如果把操作限制在对一个表进行，那么这些操作会执行得很快。然而，令人惊奇的是，这些少数的操作非常强大和重要。我们给出栈的许多应用中的三个例子，第三个例子剖析了程序是如何组织的。

#### 1. 平衡符号

编译器检查程序的语法错误，但是常常由于缺少一个符号（如遗漏一个花括号或是注释起始符）导致编译器列出上百行的错误，而真正的错误却并没有找出。

在这种情况下，一个有用的工具就是一个检验是否所有的东西都成对出现的程序。于是，每



一个右花括号、右方括号及右圆括号必然对应其相应的左半部分。 $[( )]$ 是合法的，但 $[( ])$ 就是错误的。显然，不值得为此编写一个大型程序，但这说明了这样的检验是很容易实现的。为简单起见，我们仅就圆括号、方括号和花括号进行检验并忽略出现的任何其他字符。

这个简单的算法用到一个栈，叙述如下：

做一个空栈。读入字符直至文件尾。如果字符是一个开放符号，则将其压入栈中。如果字符是一个封闭符号，那么若栈为空，则报错；若栈不为空，则将栈元素弹出。如果弹出的符号不是对应的开放符号，则报错。在文件尾，如果栈非空则报错。

96

可以确信，这个算法是可以正确运行的。很清楚，算法是线性的，事实上它只需对输入进行一次检验。因此，它是联机（on-line）的并且相当快。可以做一些附加的工作来决定当检测出错误时如何处理——例如判断可能的原因。

## 2. 后缀表达式

假设我们有一个便携计算器并想要计算一趟外出购物的花费。为此，我们将一系列数据相加并将结果乘以1.06；它是所购商品的价格加上部分商品的地方税。如果各项购物花销为4.99、5.99 和 6.99，那么输入这些数据的自然的方式将是

$$4.99 + 5.99 + 6.99 * 1.06 =$$

随着计算器的不同，这个结果或者是所要的答案19.05，或者是科学的答案18.39。最简单的四功能计算器将给出第一个答案，但是许多先进的计算器是知道乘法的优先级高于加法的。

另一方面，有些项是需要上税的而有些项则不需要上税，因此，如果只有第一项和最后一项是要上税的，那么计算的顺序

$$4.99 * 1.06 + 5.99 + 6.99 * 1.06 =$$

将在科学计算器上给出正确的答案（18.69）而在简单计算器上给出错误的答案（19.37）。科学计算器一般包含括号，因此我们总可以通过加括号的方法得到正确的答案，但是使用简单计算器我们需要记住中间结果。

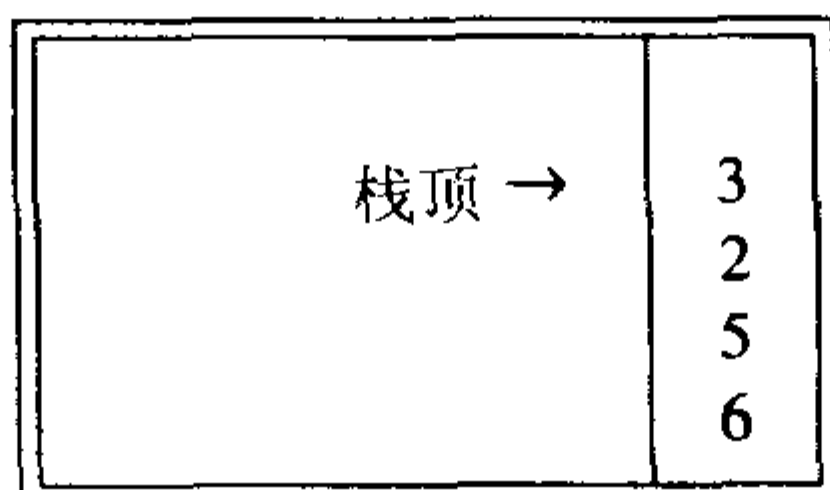
该例子的典型计算顺序是将4.99和1.06相乘并存为 $A_1$ ，然后将5.99和 $A_1$ 相加，再将结果存入 $A_1$ ；再将6.99和1.06相乘并将答案存为 $A_2$ ，最后将 $A_1$ 和 $A_2$ 相加并将最后结果放入 $A_1$ 。可以将这种操作顺序书写如下：

$$4.99 \ 1.06 * \ 5.99 + \ 6.99 \ 1.06 * +$$

这种记法叫作**后缀**（postfix）或**逆波兰记法**（reverse Polish notation），其求值过程恰好就是上面所描述的过程。计算这个问题最容易的方法是使用栈。当遇到一个数时就把它压入栈中；在遇到一个操作符时该操作符就作用于从该栈弹出的两个数（符号）上，再将所得结果压入栈中。例如，后缀表达式

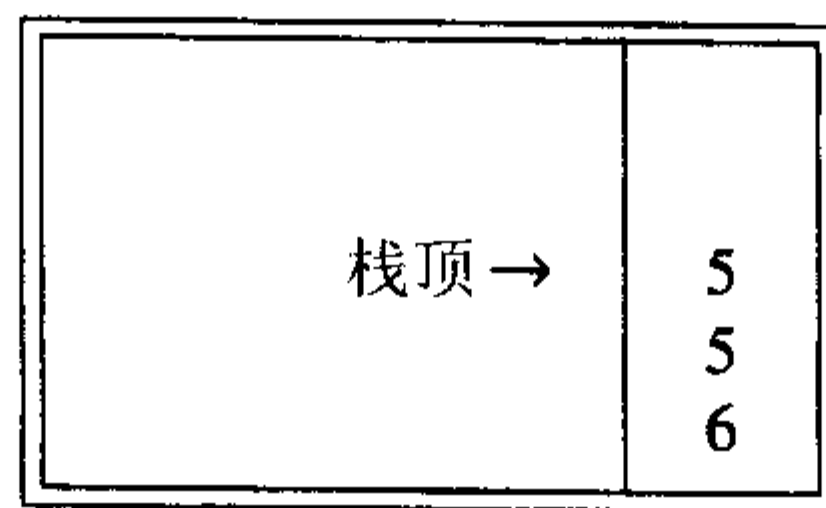
$$6 \ 5 \ 2 \ 3 + \ 8 * + \ 3 + *$$

计算如下：前4个字符放入栈中，此时栈变成

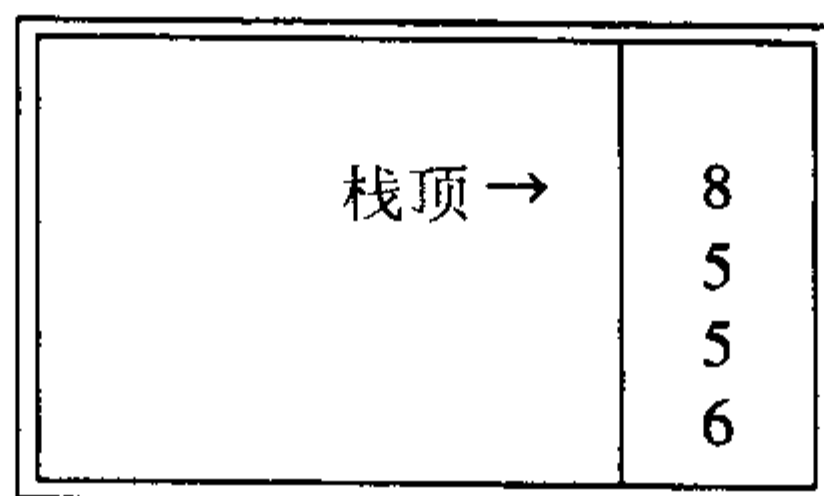


下面读到一个“+”号，所以3和2从栈中弹出并且它们的和5被压入栈中。

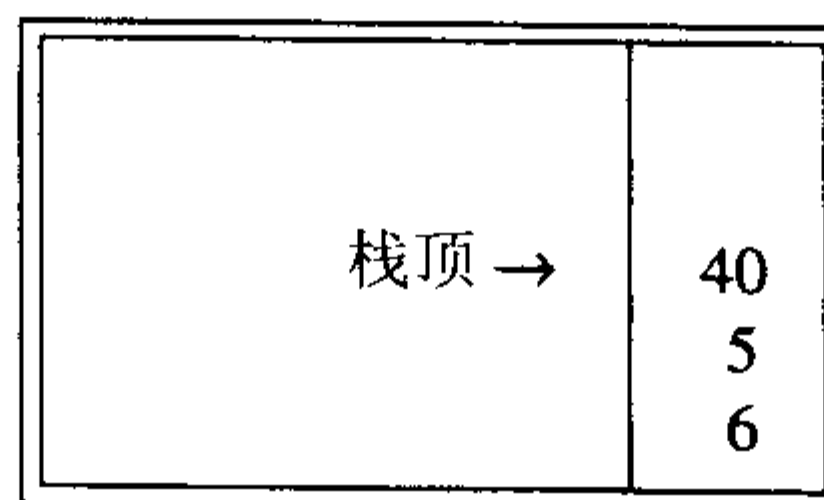
97



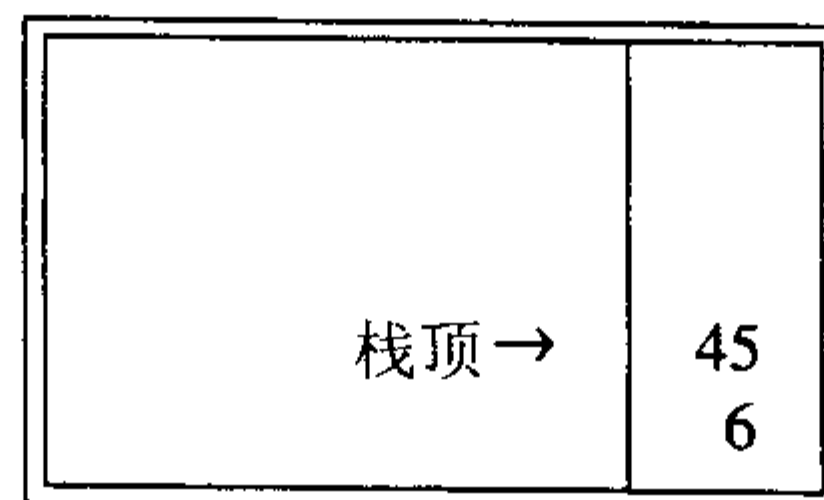
接着，8进栈。



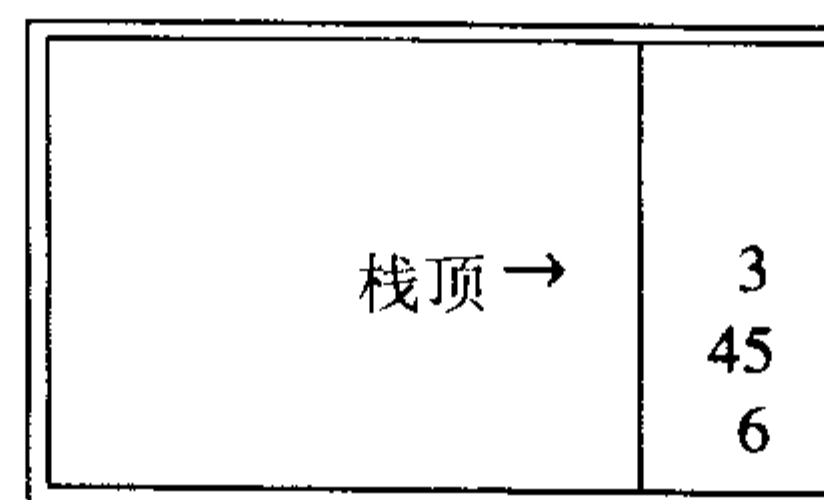
现在见到一个“\*”号，因此8和5弹出并且 $5 * 8 = 40$ 进栈。



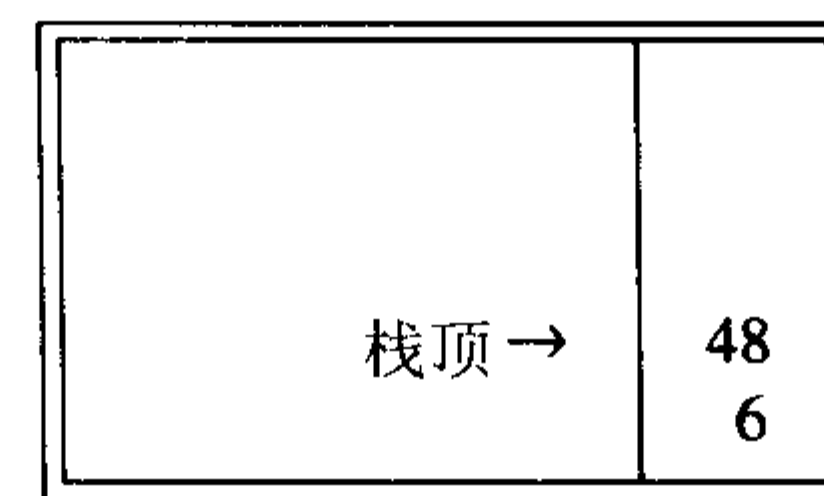
接下来又见到一个“+”号，因此40和5被弹出并且 $5 + 40 = 45$ 进栈。



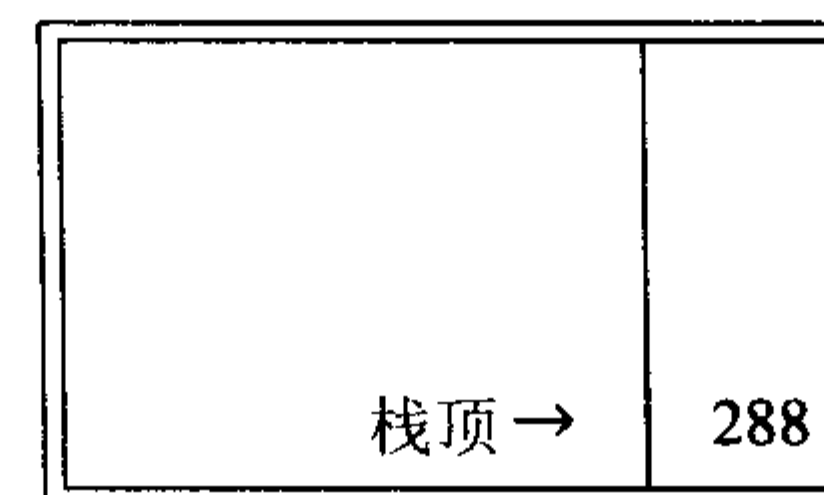
现在将3压入栈中。



98 然后“+”使得3和45从栈中弹出并将 $45 + 3 = 48$ 压入栈中。



最后，遇到一个“\*”号，从栈中弹出48和6；将结果 $6 * 48 = 288$ 压进栈中。



计算一个后缀表达式花费的时间是 $O(N)$ ，因为对输入中的每个元素的处理都是由一些栈操作

组成从而花费常数时间。该算法的计算是非常简单的。注意，当一个表达式以后缀记法给出时，没有必要知道任何优先规则；这是一个明显的优点。

3. 中缀到后缀的转换

栈不仅可以用来计算后缀表达式的值，而且还可以用来将一个标准形式的表达式（或叫作中缀式（infix））转换成后缀式。通过只允许操作符“+”、“\*”、“(”、“)”，并坚持普通的优先规则而将一般的问题浓缩成小规模的问题。还要进一步假设表达式是合法的。设欲将中缀表达式

$$a + b * c + (d * e + f) * g$$

转换成后缀表达式。正确的答案是 $a\ b\ c\ *\ +\ d\ e\ *\ f\ +\ g\ *\ +$ 。

当读到一个操作数的时候，立即把它放到输出中。操作符不立即输出，从而必须先存在某个地方。正确的做法是将已经见过的操作符放进栈中而不是放到输出中。当遇到左圆括号时我们也要将其推入栈中。计算是从一个初始化为空的栈开始。

如果见到一个右括号，那么就将栈元素弹出，将符号写出直到遇到一个（对应的）左括号，但是这个左括号只被弹出并不输出。

如果见到任何其他的符号（“+”、“\*”、“(”），那么从栈中弹出栈元素直到发现优先级更低的元素为止。有一个例外：除非是在处理“)”的时候，否则决不从栈中移走“(”。对于这种操作，“+”的优先级最低，而“(”的优先级最高。当从栈中弹出元素的工作完成后，我们再将操作符压入栈中。

最后，如果读到输入的末尾，将栈元素弹出直到该栈变成空栈，将符号写到输出中。

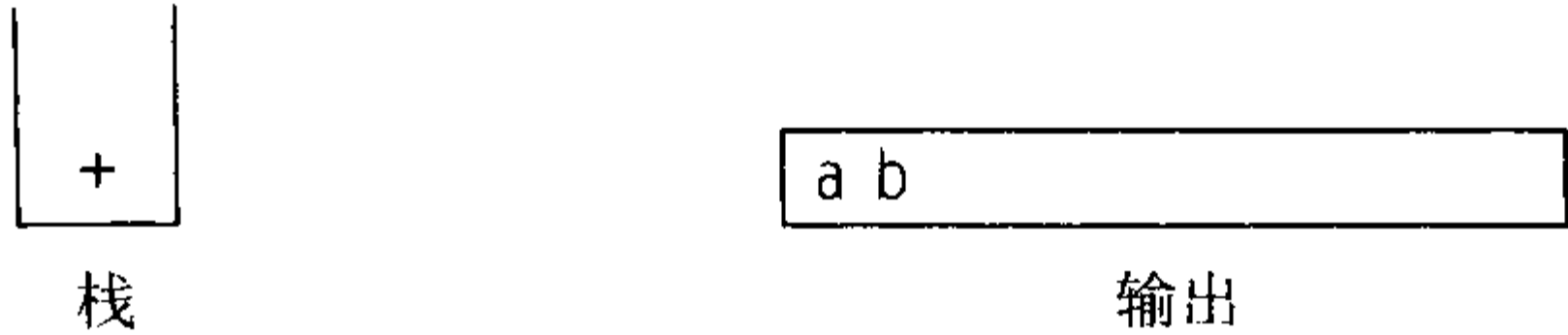
这个算法的思想是，当遇到一个操作符的时候，把它放到栈中。栈代表挂起的操作符。然而，当栈中那些具有高优先级的操作符完成使用时，就不需要再被挂起而是应该被弹出。这样，在把当前操作符放入栈中之前，那些栈中在使用当前操作符之前将要完成使用的操作符被弹出。详细的解释见下表：

99

| 表达式             | 处理第3个操作符时的栈 | 动作            |
|-----------------|-------------|---------------|
| $a * b - c + d$ | -           | -完成，+进栈       |
| $a / b + c * d$ | +           | 没有操作符完成操作，*进栈 |
| $a - b * c / d$ | - *         | *完成，/进栈       |
| $a - b * c + d$ | - *         | *和-完成，+进栈     |

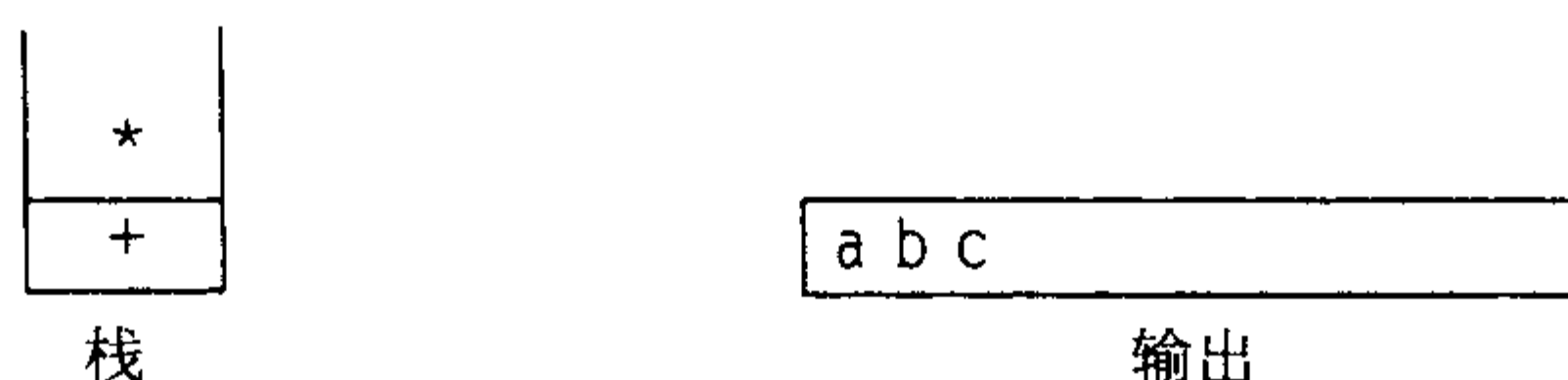
圆括号将问题变得更复杂。当左括号是一个输入符号时可以把它看成是一个高优先级的操作符（使得挂起的操作符仍是挂起的），而当它在栈中时把它看成是低优先级的操作符（从而不会被操作符意外地删除）。将右括号看成特殊的情况。

为了理解这种算法的运行机制，我们将把上面的长的中缀表达式转换成后缀形式。首先，符号a被读入，并送到输出。然后，“+”被读入并压入栈中。接下来b读入并送到输出。这时的状态如下：

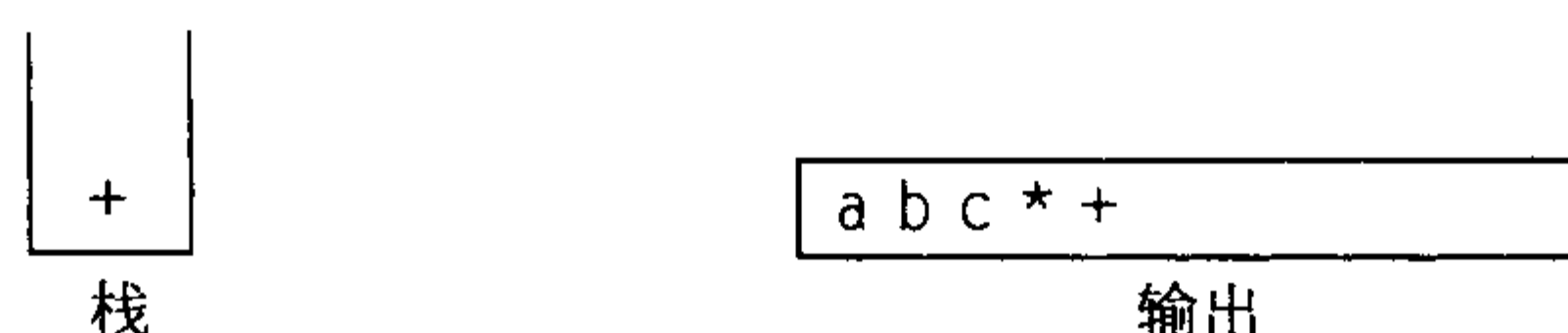


接着“\*”号读入。操作符栈的栈顶元素比“\*”的优先级低，故没有输出且“\*”进栈。接着，c被读入并输出。至此，我们有

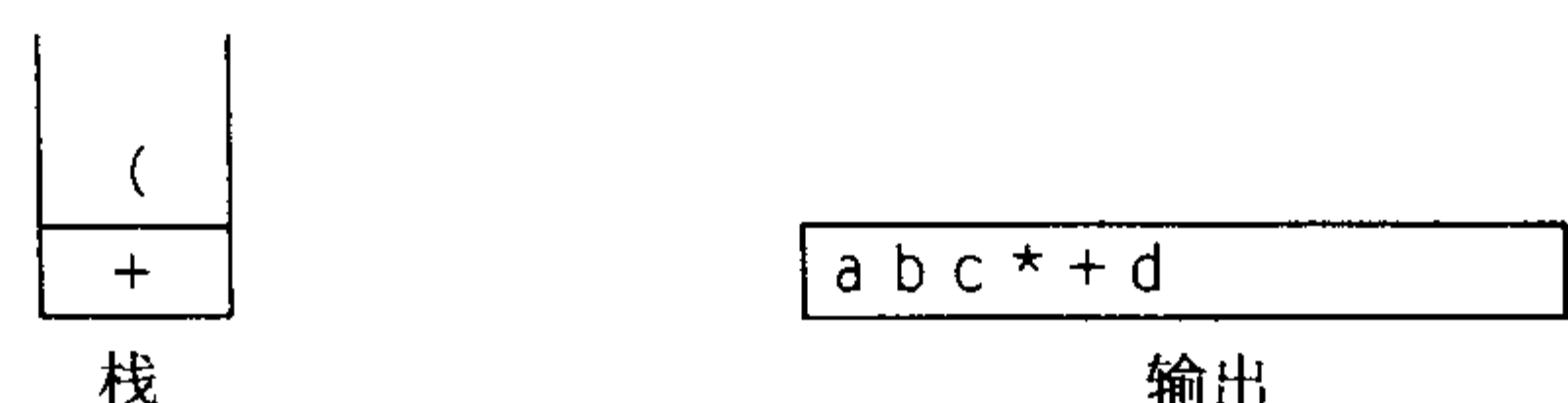




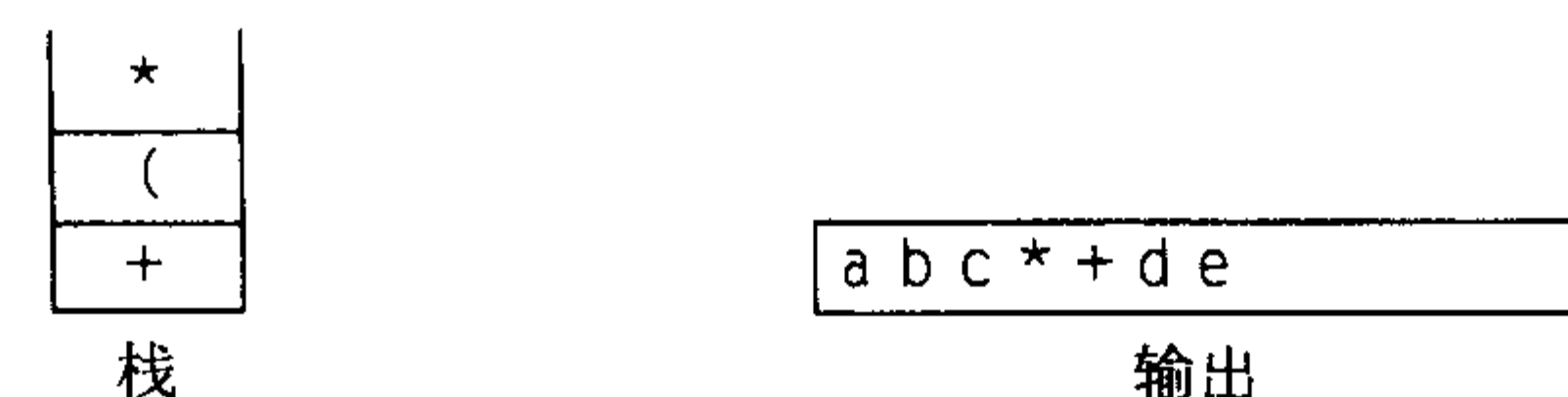
后面的符号是一个“+”号。检查一下栈可以发现，需要将“\*”从栈中弹出并放到输出中；弹出栈中剩下的“+”号，该操作符不比刚刚遇到的“+”号优先级低而是有相同的优先级；然后，将刚刚遇到的“+”号压入栈中。



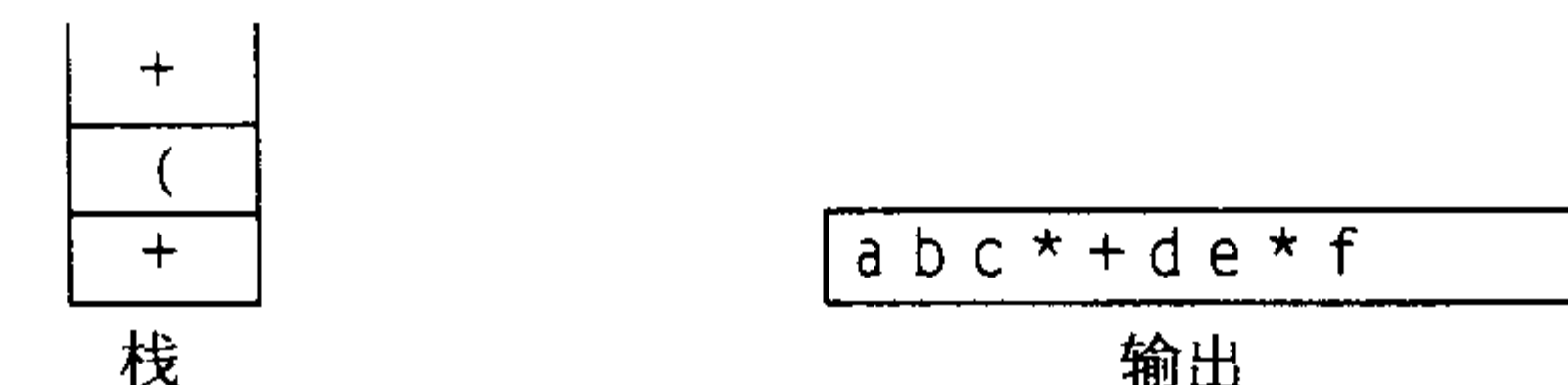
100 下一个读到的符号是一个“（”，由于有最高的优先级，因此它被放进栈中。然后，d读入并输出。



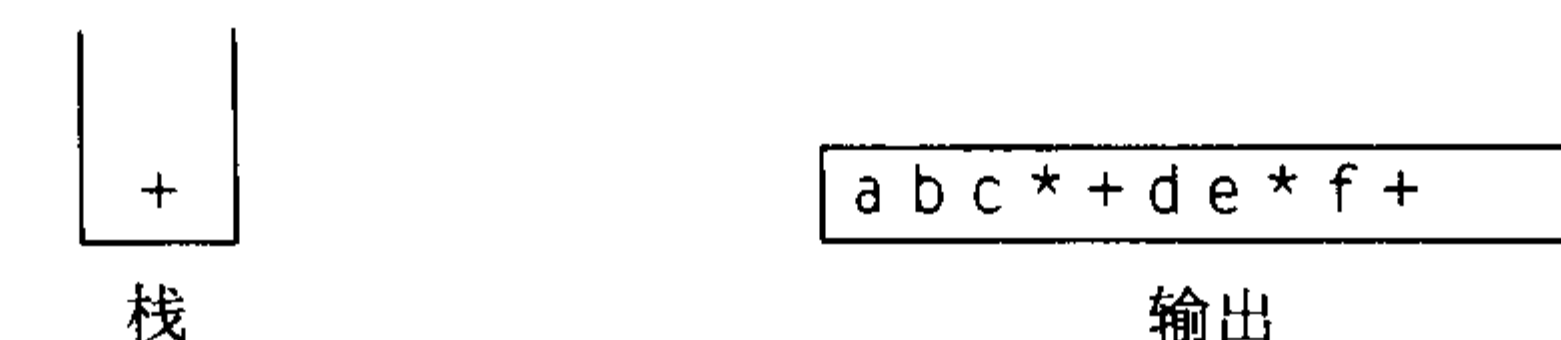
继续进行，又读到一个“\*”。由于除非正在处理闭括号，否则开括号不会从栈中弹出，因此没有输出。下一个是e，它被读入并输出。



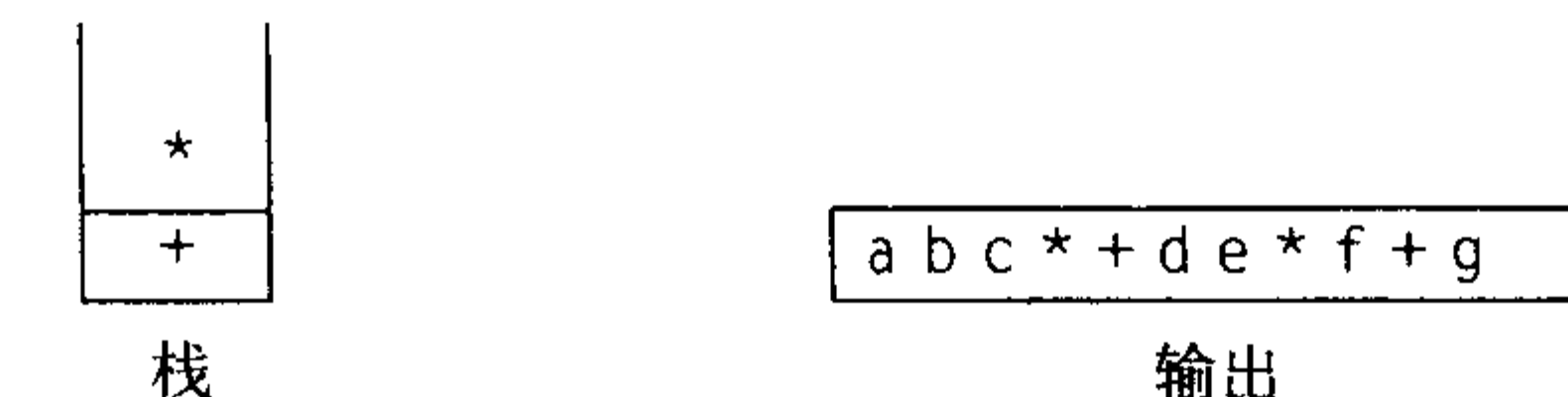
再往后读到的符号是“+”。将“\*”弹出并输出，然后将“+”压入栈中。这以后，读到f并输出。



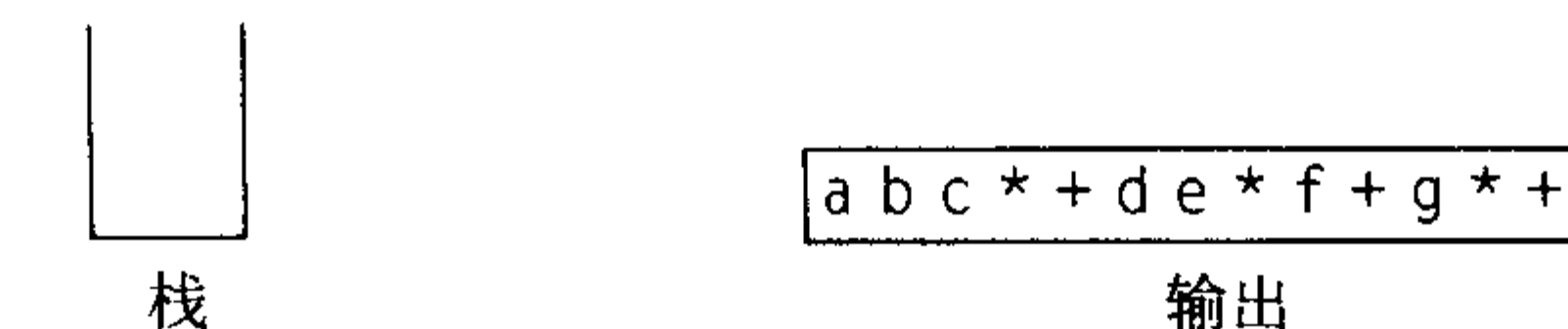
现在，读到一个“）”，因此弹出栈元素直到“（”弹出，输出一个“+”号。



下面又读到一个“\*”，该操作符被压入栈中。然后，g被读入并输出。



现在输入为空，因此我们将栈中的符号全部弹出并输出，直到栈变成空栈。



与前面相同，这种转换只需要 $O(N)$ 时间并经过一次输入后完成工作。可以通过指定减法和加

法有相同的优先级以及乘法和除法有相同的优先级而将减法和除法添加到指令系统中去。一种巧妙的想法是将表达式“a-b-c”转换成“a b - c -”而不是转换成“a b c--”。我们的算法是正确的，因为这些操作符是从左到右结合的。一般情况未必如此，比如下面的表达式就是从右到左结合的： $2^3 = 2^8 = 256$ ，而不是 $4^3 = 64$ 。我们把将取幂运算添加到操作符指令系统中的问题留做练习。

101

#### 4. 函数调用

检测平衡符号的算法提出一种在编译的过程语言和面向对象的语言中实现函数调用的方式。这里的问题是，当调用一个新函数时，主调例程的所有局部变量需要由系统存储起来，否则被调用的新函数将会重写由主调例程的变量所使用的内存。不仅如此，该主调例程的当前位置必须要存储，以便在新函数运行完后知道向哪里转移。这些变量一般由编译器指派给机器的寄存器，但存在某些冲突（通常所有的函数都得到指定给1#寄存器的某些变量），特别是涉及递归的时候。该问题类似于平衡符号问题的原因在于，函数调用和函数返回基本上类似于开括号和闭括号，二者想法是一样的。

当存在函数调用的时候，需要存储的所有重要信息，诸如寄存器的值（对应变量的名字）和返回地址（它可从程序计数器得到，一般情况下是在寄存器中）等，都要以抽象的方式存在“一张纸上”并被置于一个堆（pile）的顶部。然后控制转移到新函数，该函数自由地用它的值代替这些寄存器。如果它又进行其他的函数调用，那么也遵循相同的过程。当该函数要返回时，它查看堆顶部的那张“纸”并复原所有的寄存器。然后它进行返回转移。

显然，所有工作均可由一个栈来完成，而这正是在实现递归的每一种程序设计语言中实际发生的事实。所存储的信息或称为活动记录（activation record），或称为栈帧（stack frame）。一般情况下，需要做些微调：当前环境是由栈顶描述的。因此，一条返回语句就可给出前面的运行环境（不用复制）。在实际计算机中，栈常常是从内存分区的高端向下增长，而在许多系统中是不检测溢出的。由于同时有太多的正在运行着的函数，因此用尽栈空间的情况总是可能发生的。毋庸置疑，用尽栈空间总是致命的错误。

在不进行栈溢出检测的语言和系统中，程序将会没有明确说明地崩溃。

在正常情况下不应该用尽栈空间；发生这种情况通常意味着有失控递归（忘记基准情形）的存在。另一方面，某些完全合法并且表面上无害的程序也可能用尽栈空间。图3-25中的例程打印一个容器，该例程完全合法而且事实上也是正确的。它正确地处理空容器的基准情形，并且递归也没问题。可以证明这个程序是正确的。但是，如果这个链表含有20 000个元素要打印，那么就存在表示第11行嵌套调用的20 000个活动记录的一个栈。一般这些活动记录由于它们包含全部信息而特别庞大，因此这个程序很可能要用尽栈空间（如果20 000个元素还不足以使程序崩溃，那么可用更大的数字代替）。

```

1  /**
2   * Print container from start up to but not including end.
3   */
4  template <typename Iterator>
5  void print( Iterator start, Iterator end, ostream & out = cout )
6  {
7      if( start == end )
8          return;
9
10     out << *start++ << endl;    // Print and advance start
11     print( start, end, out );
12 }
```

图3-25 递归的不当使用：打印一个容器

102

这个程序称为尾递归 (tail recursion)，是极差的使用递归的例子。尾递归指的是在最后一行的递归调用。通过将代码放到一个while循环中并用每个函数的一个参数代替递归调用，可以机械地消除尾递归。这模拟了递归调用，因为什么也不需要存储；在递归调用结束之后，实际上没有必要知道存储的值。因此，我们就可以带着在一次递归调用中已经用过的那些值转移到函数的顶部。图3-26中的函数显示了通过这种算法机械地改进后的程序。尾递归的去除非常简单，某些编译器能够自动地完成。即使如此，最好还是不要让程序带着尾递归。

```

1  /**
2   * Print container from start up to but not including end.
3   */
4  template <typename Iterator>
5  void print( Iterator start, Iterator end, ostream & out = cout )
6  {
7      while( true )
8      {
9          if( start == end )
10             return;
11
12         out << *start++ << endl;    // Print and advance start
13     }
14 }

```

图3-26 不使用递归打印一个容器；编译器可以完成这项工作（最好不这样做）

103

递归总能够被彻底除去（编译器在转变成汇编语言时完成递归的去除），但是这么做是相当冗长乏味的。一般方法是要求使用一个栈，而且仅当你能够把最低限度的最小值放到栈上时，这个方法才值得一用。这里不对此做进一步的详细讨论，只是指出，虽然非递归程序一般说来确实比等价的递归程序要快，但是速度优势的代价却是由于去除递归而使得程序的清晰性受到了影响。

## 3.7 队列ADT

像栈一样，队列 (queue) 也是表。然而，使用队列时插入是在一端进行的，而删除则是在另一端进行的。

### 3.7.1 队列模型

队列的基本操作是enqueue（入队），它是在表的末端（称为队尾）插入一个元素；dequeue（出队），它是删除（并返回）表的开头（叫作队头）的元素。图3-27显示了一个队列的抽象模型。

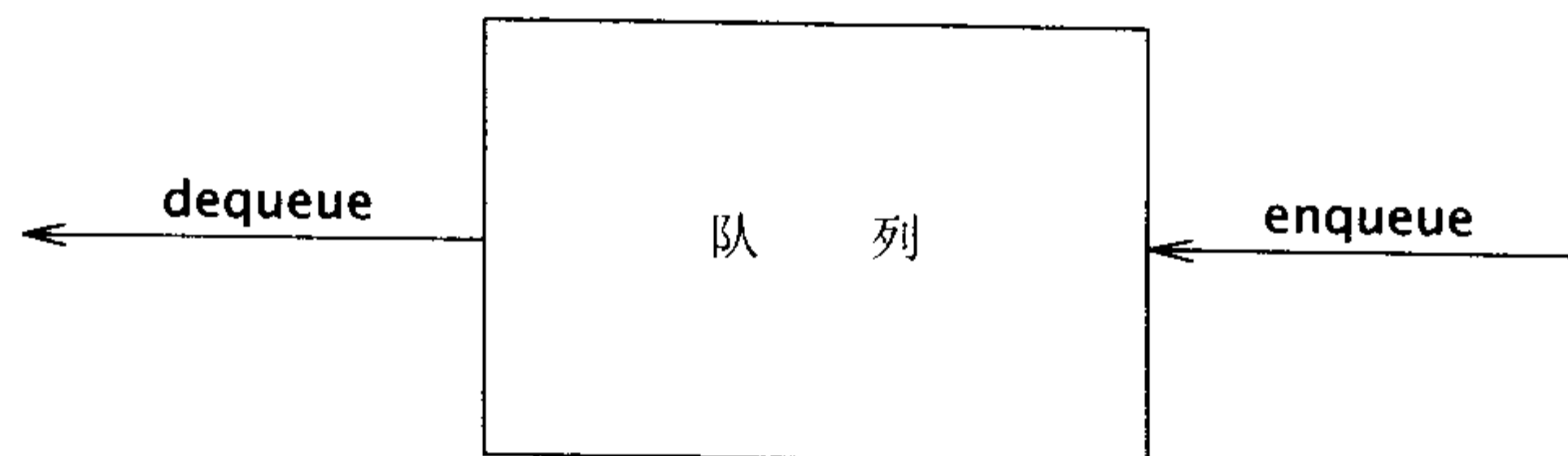


图3-27 队列模型

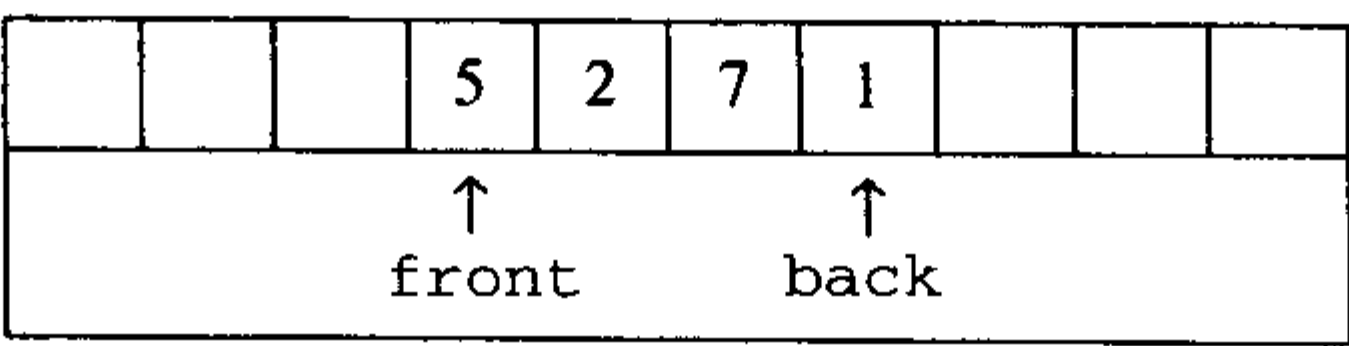
### 3.7.2 队列的数组实现

如同栈的情形一样，对于队列而言任何表的实现都是合法的。像栈一样，对于每一种操作，链表实现和数组实现都给出快速的 $O(1)$ 运行时间。队列的链表实现非常直观，留作练习。现在我



们讨论队列的数组实现。

对于每一个队列数据结构，我们保留一个数组theArray以及位置front和back，它们代表队列的两端。我们还要记录实际存在于队列中的元素的个数currentSize。下图表示处于某个中间状态的一个队列。

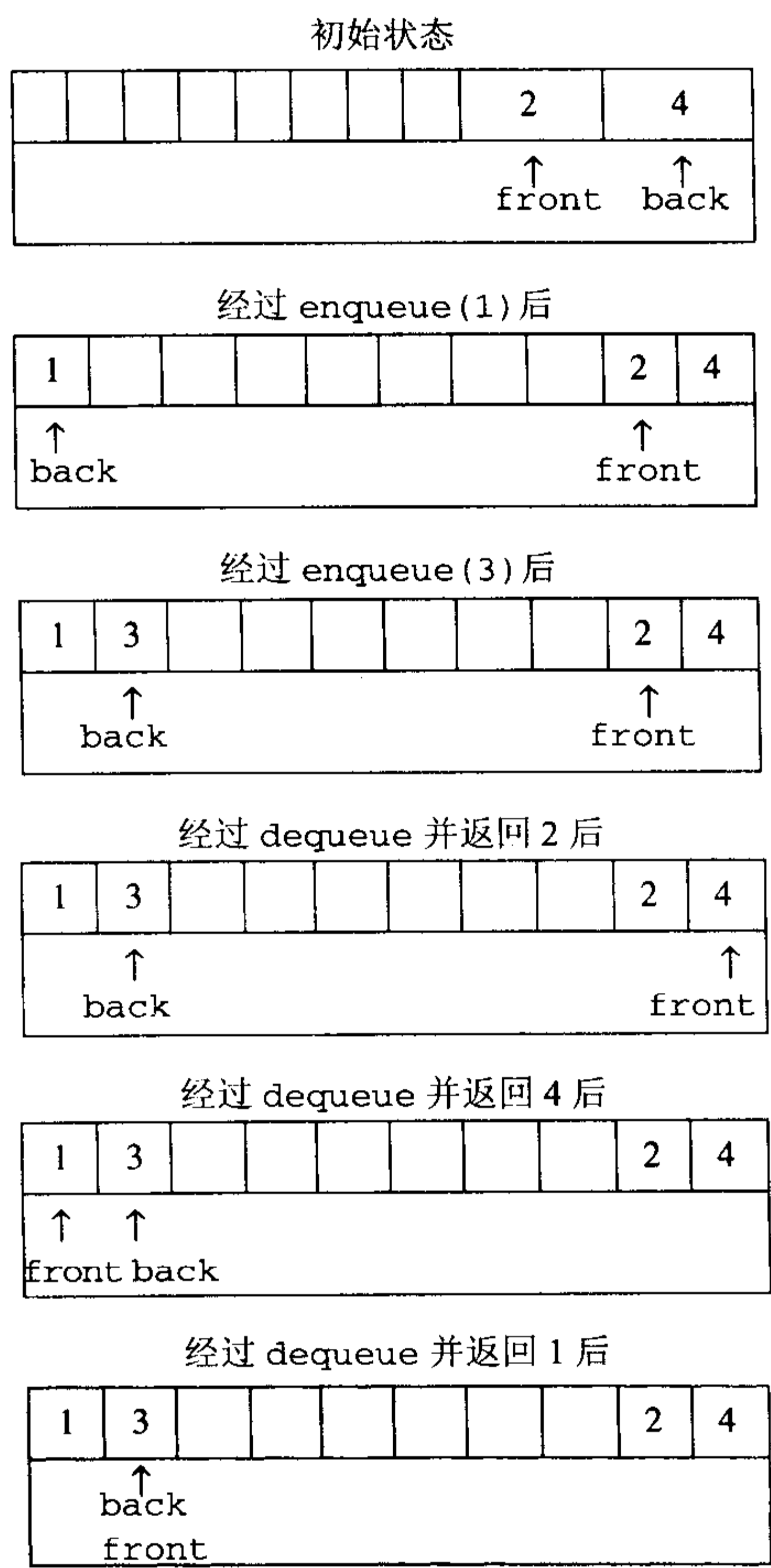


操作应该是清楚的。要 enqueue 元素 x，可将 currentSize 和 back 增 1，然后置 theArray[back]=x。要 dequeue 一个元素，可以置返回值为 theArray[front]，将 currentSize 减 1，再将 front 增 1。其他的方法也可以使用（将在后面讨论）。现在论述错误的检测。

104

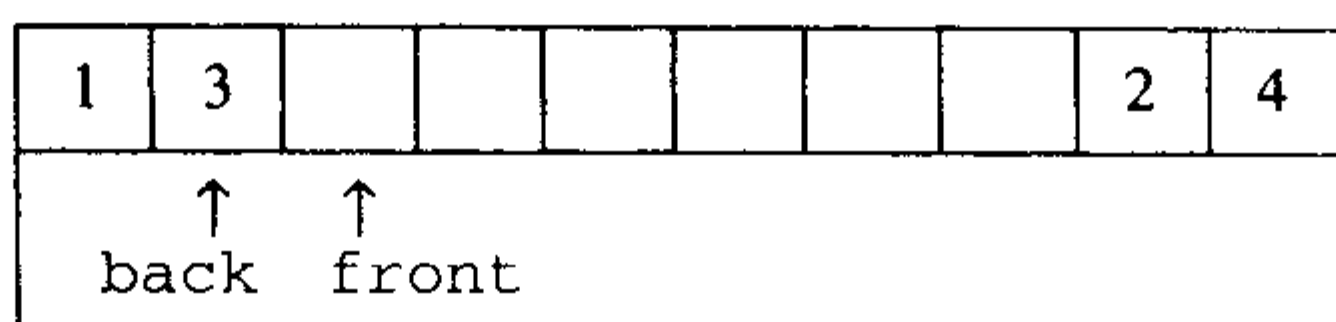
这种实现存在一个潜在的问题。经过 10 次 enqueue 后，队列似乎是满了，因为 back 现在是数组的最后一个下标，而下一次执行 enqueue 就会是一个不存在的位置。然而，队列中也许只存在几个元素，因为若干元素可能已经出队了。像栈一样，即使在有许多操作的情况下队列也常常不是很大。

简单的解决方法是，只要 front 或 back 到达数组的尾端，就绕回到开头。下列图显示了在某些操作期间的队列情况。这称为循环数组（circular array）实现。



105

经过 dequeue 并返回 3 后，同时使队列为空



实现回绕所需要的附加代码是极小的（虽然它可能使得运行时间加倍）。如果 front 或 back 增1使得它超越了数组，那么其值就要重置为数组的第一个位置。

有些程序设计员使用不同的方法表示队列的队头和队尾。例如，有人不使用一项来记录大小，因为他们依靠当队列为空（back=front-1）时的基准情形。队列的大小通过比较 back 和 front 隐式地算出。这是一种非常隐秘的方法，因为存在某些特殊的情形，因此，如果想修改用这种方法编写的程序，那就要特别地小心。如果 currentSize 不作为显式的数据成员被保留，那么当存在 theArray.length-1 个元素时队列就满了，因为只有 theArray.length 个不同的大小可被区分，而 0 是其中的一个。可以采用任意一种你喜欢的风格，但要确保所有例程都是一致的。由于实现方法有多种选择，因此如果不使用 currentSize 数据成员，那就可能有必要进行一些注释。

在确定使用 enqueue 的次数不会大于队列的大小的应用中，使用回绕是没有必要的。像栈一样，除非主调例程肯定队列非空，否则 dequeue 很少执行。因此对这种操作，只要不是关键的代码，错误检查常常被跳过。一般说来这样做并不合适，因为节省的时间是极少的。

### 3.7.3 队列的应用

有许多使用队列得到高效运行时间的算法。它们当中有些可以在图论中找到，我们将在第9章讨论它们。这里，先给出一些应用队列的简单例子。

106

当作业送交给一台打印机的时候，它们就以到达的顺序被排列起来。因此，被送往行式打印机的作业基本上是被放到一个队列中。<sup>1</sup>

事实上每一个实际生活中的排队都（应该）是一个队列。例如，在售票口排列的队是队列，因为服务的顺序是先到的先买票。

另一个例子是关于计算机网络的。有许多种 PC 机的网络设置，其中磁盘是放在一台叫作文件服务器（file server）的机器上的。使用其他计算机的用户是按照先到先使用的原则访问文件的，因此其数据结构是一个队列。

进一步的例子如下：

- 当所有的接线员忙不开的时候，对大公司的传呼一般都被放到一个队列中。
- 在资源有限的大型大学里，如果所有的终端都被占用，学生们就必须在一个等待表上签字登记。在终端上呆得时间最长的学生将首先被强制离开，而等待时间最长的学生则将是下一个被允许使用终端的用户。

一个称为排队论（queuing theory）的数学分支用概率的方法处理一些诸如计算用户预计要排队等待的时间、等待的队伍能够排多长等类问题。问题的答案依赖于用户加入队列的频率以及一旦用户得到服务时处理服务花费的时间。这两个参数作为概率分布函数给出。在一些简单的情况下，答案可以解析地算出。一种简单的例子是一条电话线有一个接线员。如果接线员忙，打来的电话就被放到一个等待队列中（这还与某个容许的最大限度有关）。这个问题在商业上很重要，因为研究表明，人们会很快挂上电话。

1. 我们说基本上是因为作业可以被除去。这等于从队列的中间进行的一次删除，它违反了队列的严格定义。

如果有 $k$ 个接线员，那么这个问题解决起来要困难得多。解析求解困难的问题往往使用模拟的方法求解。此时，需要使用一个队列来进行模拟。如果 $k$ 很大，那么还需要其他一些数据结构来使得模拟更有效地进行。在第6章将会看到模拟是如何进行的。那时我们将对几个不同的 $k$ 值进行模拟并选择能够给出合理等待时间的最小的 $k$ 。

正如栈一样，队列还有其他丰富的用途，这样一种简单的数据结构竟然能够如此重要，实在令人吃惊。

## 小结

本章描述了一些ADT的概念，并且利用三种最常见的抽象数据类型阐述了这个概念。主要目的就是为将抽象数据类型的实现与它们的功能分开。程序必须知道操作都做些什么，但是如果不知道如何做就更好。

在所有的计算机科学中，表、栈和队列或许都是三个基本的数据结构，大量的例子证明了它们广泛的用途。特别地，我们看到栈是如何用来记录函数调用以及如何真实地实现递归的。这对于理解是非常重要的，其原因不只是它使得过程语言成为可能，而且还因为知道递归的实现而揭开了围绕其使用的神秘面纱。虽然递归非常强大，但是它并不是完全随意的操作；递归的误用和乱用可能导致程序崩溃。

107

## 练习

- 3.1 给定一个链表 $L$ 和另一个链表 $P$ ，它们包含以升序排列的整数。操作`printLots(L, P)`将打印 $L$ 中那些由 $P$ 所指定的位置上的元素。例如，如果 $P = 1, 3, 4, 6$ ，那么 $L$ 中的第1、3、4和6个元素被打印出来。写出过程`printLots(L, P)`。只可以使用公有的STL容器操作。该过程的运行时间是多少？
- 3.2 通过只调整链（而不是数据）来交换两个相邻的元素，使用：
  - a. 单向链表。
  - b. 双向链表。
- 3.3 实现STL的`find`例程。该例程返回的`iterator`包含从`start`开始一直到`end`（不包含`end`）的范围内第一个出现的 $x$ 。如果 $x$ 没有找到，就返回`end`。该非类（全局函数）的签名如下：
 

```
template <typename Iterator, typename Object>
iterator find( Iterator start, Iterator end, const Object & x )
```
- 3.4 给定两个排序后的表 $L_1$ 和 $L_2$ 。写出一个程序仅使用基本的表操作来计算 $L_1 \cap L_2$ 。
- 3.5 给定两个排序后的表 $L_1$ 和 $L_2$ 。写出一个程序仅使用基本的表操作来计算 $L_1 \cup L_2$ 。
- 3.6 Josephus问题是下面的这个游戏：有 $N$ 个人坐成一圈，编号为1至 $N$ 。从编号为1的人开始传递热马铃薯。 $M$ 次传递之后，持有热马铃薯的人退出游戏，圈缩小，然后游戏从退出人下面的人开始，继续进行。最终留下来的人获胜。这样，如果 $M=0$ 并且 $N=5$ ，那么参加游戏的人依次退出，5号获胜。如果 $M=1$ 并且 $N=5$ ，那么退出的顺序就是2、4、1、5。
  - a. 写出一个程序来解决Josephus问题，此时 $M$ 和 $N$ 为任意值。尽可能使程序高效，同时确保存储单元被正确处理。
  - b. 程序的运行时间是多少？
  - c. 当 $M=1$ 时，程序的运行时间是多少？对于 $N$ 的较大值（ $N > 100\,000$ ），`delete`例程对程序运行速度的影响有多大？



- 3.7 修改Vector类，添加索引时的边界检测功能。
- 3.8 给Vector类添加insert和erase。
- 3.9 对于vector，按照C++标准，对push\_back、pop\_back、inset或erase的调用将使所有的指向vector的迭代器失效（潜在生成废物的可能）。为什么？
- 3.10 修改Vector类，通过赋予迭代器类类型而不是指针变量，来提供严格的迭代器检验。正如练习3.9所描述的一样，最困难的部分是处理失效的迭代器。
- 3.11 假设一个单向链表的实现有一个表头结点，但是没有尾结点，并且只有一个指向表头结点的指针。

108

写一个类，使之包括的方法可以

- a. 返回链表的大小。
  - b. 打印链表。
  - c. 检测值x是否在链表中。
  - d. 如果值x没在链表中，则将其加入链表。
  - e. 如果值x包含在链表中，则删除这个值。
- 3.12 重复练习3.11，保持单向链表总是处于排序状态。
- 3.13 给List迭代器类添加对operator--的支持。
- 3.14 读STL迭代器的值需要使用operator++操作，该操作依次推进迭代器。在某些情况下，读表中下一项的值而不推进迭代器也许更好。写出如下声明的成员函数来实现一般情况下的这个功能。

```
const_iterator operator+( int k ) const;
```

二元操作符operator+返回一个对应于当前位置前面第k个位置的迭代器。

- 3.15 给List类添加splice操作。其声明如下：

```
void splice( iterator position, List<T>& lst);
```

删除lst中的所有项，并将这些项放在List \*this中的位置position之前。lst和\*this必须是不同的表。所写的例程必须是常量时间的。

- 3.16 给STL List类实现添加逆向迭代器。定义reverse\_iterator和const\_reverse\_iterator。添加方法rbegin和rend，来分别返回逆向迭代器所指向的末尾标记前的项的位置和表头结点的位置。逆向迭代器在内部逆转++和--操作符的意义。应该可以使用下面的代码逆向打印表L

```
List<Object>::reverse_iterator itr = L.rbegin( );
while( itr!= L.rend())
    cout << *itr++ << endl;
```

- 3.17 使用3.5节末尾建议的思想修改List类，以提供严格的迭代器检查功能。
- 3.18 对List使用erase方法时，所有的指向被删除结点的迭代器iterator都失效。这样的迭代器称为**废物**（stale）。描述一个高效的算法，该算法必须保证所有的针对废物迭代器的操作都将迭代器的当前值当做NULL处理。注意，可能会有很多的废物迭代器。你必须解释哪一个类需要重写，以实现你的算法。
- 3.19 重写List类，要求不使用表头结点和尾结点。说明该类与3.5节提供的类的区别。
- 3.20 不同于我们已经给出的删除方法，另一种是使用**懒惰删除**（lazy deletion）的方法。为删除一个元素，我们只是标记该元素被删除（使用一个附加的位字段）。表中被删除和未被删除元素的个数作为数据结构的一部分被保留。如果被删除元素和未被删除元素一样多，则遍历整个表，对所有被标记的结点执行标准的删除算法。

109

- a. 列出懒惰删除的优点和缺点。
- b. 写出使用懒惰删除实现标准链表操作的例程。

- 3.21 编写检测下列语言中平衡符号的程序：
- Pascal (begin/end, ( ), [ ], {}).
  - C++ (/ \* \*/ , ( ), [ ], {}).
- \*c. 解释如何打印出反映可能的出错原因的出错信息。
- 3.22 编写一个程序计算后缀表达式的值。
- 3.23
- 编写一个程序将中缀表达式转换成后缀表达式，该中缀表达式包含 “(”、“)”、“+”、“-”、“\*”和“/”。
  - 把幂操作符添加到你的指令系统中去。
  - 编写一个程序将后缀表达式转换成中缀表达式。
- 3.24 编写仅用一个数组而实现两个栈的例程。除非数组的每一个单元都被使用，否则栈例程不能有溢出声明。
- 3.25
- 提出支持栈的push和pop操作以及第三种操作findMin的数据结构，其中findMin返回该数据结构的最小元素，所有操作在最坏的情况下的运行时间都是 $O(1)$ 。
  - 证明，如果我们加入第四种操作deleteMin，那么至少有一种操作必然花费 $\Omega(\log N)$ 时间，其中，deleteMin是找出并删除最小的元素（本题需要阅读第7章）。
- \*3.26 指出如何用—一个数组实现三个栈。
- 3.27 在2.4节中用于计算斐波那契数的递归例程如果在 $N=50$ 下运行，栈空间有可能用完吗？为什么？
- 3.28 **双端队列（deque）**是由某些项的一个表组成的数据结构，对该数据结构可以进行下列操作：
- push(x)：将项x插入到双端队列的前端。
- pop()：从双端队列中删除前端项并将其返回。
- inject(x)：将项x插入到双端队列的尾端。
- eject()：从双端队列中删除尾端项并将其返回。
- 编写支持双端队列的例程，每种操作均花费 $O(1)$ 时间。
- 3.29 编写一个逆向打印单向链表的算法，要求只使用固定的额外空间。不可以使用递归但是可以假定算法是一个表成员函数。如果例程是常量成员函数的话，算法还可以实现吗？
- 3.30
- 写出一个自调整表的数组实现。在**自调整表（self-adjusting list）**中，所有的插入操作都发生在表的前端。自调整表添加了一个find操作，当一个元素由find访问的时候，该元素就被移到表的前端，而其他元素的相对顺序保持不变。
  - 写出一个自调整表的链表实现。
- \*c. 假设每一个元素被访问的概率 $P_i$ 都是固定的。说明最高访问概率的元素应该在表的最前面。
- 3.31 使用单向链表高效地实现栈类。要求不含表头结点或尾结点。
- 3.32 使用单向链表高效地实现队列类。要求不含表头结点或尾结点。
- 3.33 使用循环数组高效地实现队列类。可以使用vector（而不是基本数组）作为基本的数组结构。
- 3.34 如果从某个结点 $p$ 开始，接着跟有足够数目的next链将我们带回到结点 $p$ ，那么这个链表包含一个循环。 $p$ 不必是该表的第一个结点。假设给你一个链表，它包含 $N$ 个结点；不过 $N$ 的值是不知道的。
- 设计一个 $O(N)$ 算法来确定一个表中是否包含一个循环。可以使用 $O(N)$ 的额外空间。
- \*b. 重复a部分，但是仅使用 $O(1)$ 的额外空间。（提示：使用两个迭代器，它们最初在表的开始处，但以不同的速度推进。）
- 3.35 实现队列的一种方法是使用一个**循环链表**。在循环链表（circular linked list）中，最后一个结点的next指针指向第一个结点。假设该表不包含表头，而且我们最多可以保留一个迭代器，它对应表中的一个结点。对于下列的哪种表示，所有的基本队列操作可以以常数最坏情形时间执行？证明你的答案是正确的。

- a. 保留一个迭代器，它对应该表的第一项。
  - b. 保留一个迭代器，它对应该表的最后一项。
- 3.36 设我们有指向单向链表的一个结点的指针，而且保证该结点不是该表的最后结点。我们没有指向任何其他结点的指针（后面的链除外）。描述一个 $O(1)$ 算法，该算法逻辑上删除存储在该链表这样一个结点上的值，同时保持链表的完整性（提示：涉及下一个结点）。
- 3.37 设单向链表用头结点和尾结点实现。描述常数时间算法以
- a. 在位置 $p$ （由一个迭代器给出）前插入一项 $x$ 。
  - b. 删除存储在位置 $p$ （由一个迭代器给出）的项。



## 树

对于大量的输入数据，链表的线性访问时间太长，不宜使用。本章我们介绍一种简单的数据结构，其大部分操作的运行时间平均为 $O(\log N)$ 。我们还将简述对这种数据结构在概念上的简单的修改，它保证了在最坏情形下的上述时间界。此外，还讨论了第二种修改，对于长的指令序列它对每种操作的运行时间基本上是 $O(\log N)$ 。

我们涉及的这种数据结构叫作**二叉查找树**（binary search tree）。二叉查找树是在很多应用程序中都有使用的两个库集合类set和map的实现基础。在计算机科学中树（tree）是非常有用的抽象概念，因此，我们将讨论树在其他更一般的应用中的使用。在这一章，我们将

- 了解树是如何用于实现几个流行的操作系统中的文件系统的。
- 了解树如何用来计算算术表达式的值。
- 指出如何利用树支持以 $O(\log N)$ 平均时间进行的各种搜索操作，以及如何细化以得到最坏情况时间界 $O(\log N)$ 。我们还将讨论当数据被存放在磁盘上时如何实现这些操作。
- 讨论并使用set和map类。

## 4.1 预备知识

树（tree）可以用几种方式定义。定义树的一种自然的方式是递归的方法。一棵树是一些结点的集合。这个集合可以是空集；若不是空集，则树由称作**根**（root）的结点 $r$ 以及零个或多个非空的（子）树 $T_1, T_2, \dots, T_k$ 组成，这些子树中每一棵的根都被来自根 $r$ 的一条有向的边（edge）所连接。

每一棵子树的根叫作根 $r$ 的**儿子**（child），而 $r$ 是每一棵子树的根的父亲（parent）。图4-1显示了用递归定义的典型的树。

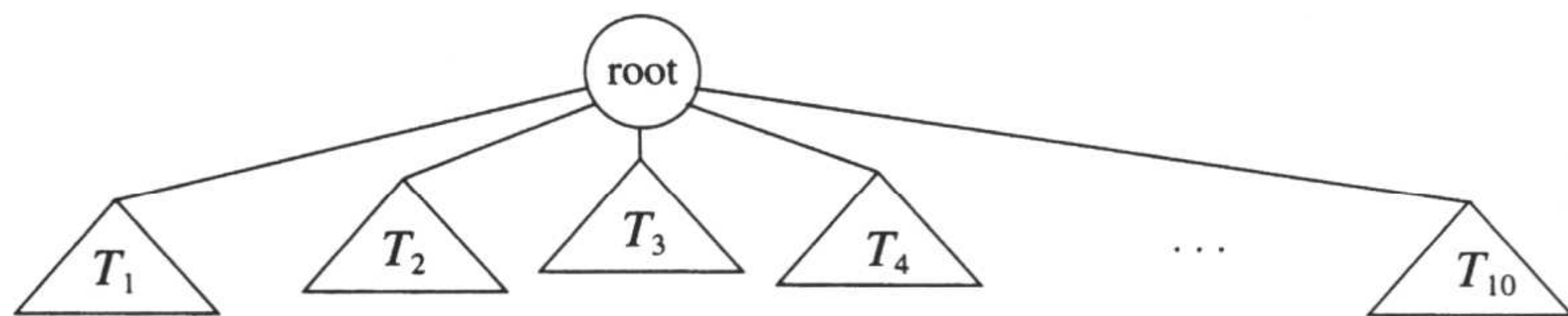


图4-1 一般的树

从递归定义中可以发现，一棵树是 $N$ 个结点和 $N-1$ 条边的集合，其中的一个结点叫作根。存在 $N-1$ 条边的结论是由下面的事实得出的：每条边都将某个结点连接到它的父亲，而除去根结点外每一个结点都有一个父亲（见图4-2）。

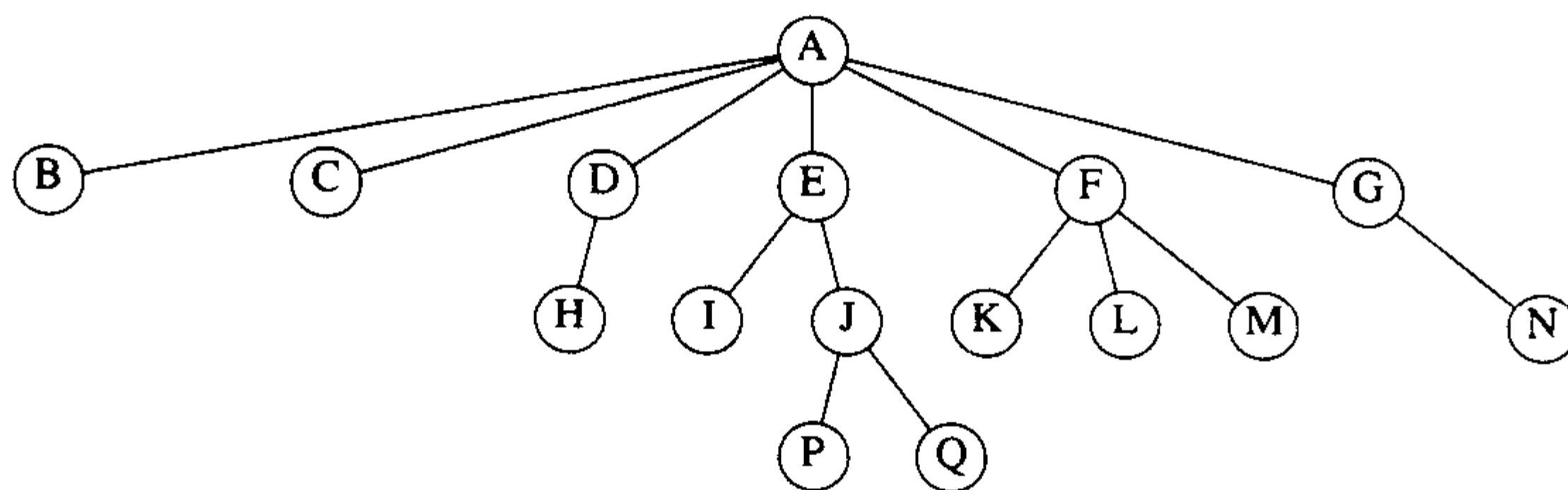


图4-2 一棵具体的树

113 在图4-2的树中，结点A是根。结点F有一个父亲A并且有儿子K、L和M。每一个结点可以有任意多个儿子，也可能没有儿子。没有儿子的结点称为叶（leaf）结点；图4-2中的叶结点（树叶）是B、C、H、I、P、Q、K、L、M和N。具有相同父亲的结点为兄弟（siblings）结点；因此，K、L和M都是兄弟。用类似的方法可以定义祖父（grandparent）和孙子（grandchild）关系。

从结点 $n_1$ 到 $n_k$ 的路径（path）定义为结点 $n_1, n_2, \dots, n_k$ 的一个序列，使得对于 $1 \leq i < k$ ，结点 $n_i$ 是 $n_{i+1}$ 的父亲。路径的长（length）为路径上的边的条数，即 $k-1$ 。从每一个结点到它自己有一条长为0的路径。注意，在一棵树中从根到每个结点恰好存在一条路径。

对任意结点 $n_i$ ， $n_i$ 的深度（depth）为从根到 $n_i$ 的唯一路径的长。因此，根的深度为0。 $n_i$ 的高（height）是从 $n_i$ 到一片树叶的最长路径的长。因此所有的树叶的高都是0。一棵树的高等于它的根的高。对于图4-2中的树，E的深度为1而高为2；F的深度为1而高也是1；该树的高为3。一个树的深度等于它的最深的树叶的深度；该深度总是等于这棵树的高。

如果存在从 $n_1$ 到 $n_2$ 的一条路径，那么 $n_1$ 是 $n_2$ 的一位祖先（ancestor）而 $n_2$ 是 $n_1$ 的一个后裔（descendant）。如果 $n_1 \neq n_2$ ，那么 $n_1$ 是 $n_2$ 的一位真祖先（proper ancestor）而 $n_2$ 是 $n_1$ 的一个真后裔（proper descendant）。

#### 4.1.1 树的实现

114 实现树的一种方法是在每一个结点除数据外还要有一些链，来指向该结点的每一个儿子。然而，由于每个结点的儿子数可能变化很大并且事先不知道，因此在数据结构中建立到各儿子结点的直接链接是不可行的，因为这样会产生太多浪费的空间。实际上解法很简单：将每个结点的所有儿子都放在树结点的链表中。图4-3所示是非常典型的声明。

```

1 struct TreeNode
2 {
3     Object    element;
4     TreeNode *firstChild;
5     TreeNode *nextSibling;
6 };
  
```

图4-3 树结点的声明

图4-4显示了一棵树是如何用这种实现方法表示出来的。图中向下的箭头是指向firstChild的链。从左到右的箭头是指向nextSibling的链。因为空链太多，所以没有把它们画出。

在图4-4所示的树中，结点E有一个链指向兄弟（F），另一链指向儿子（I），而有的结点两种链都没有。



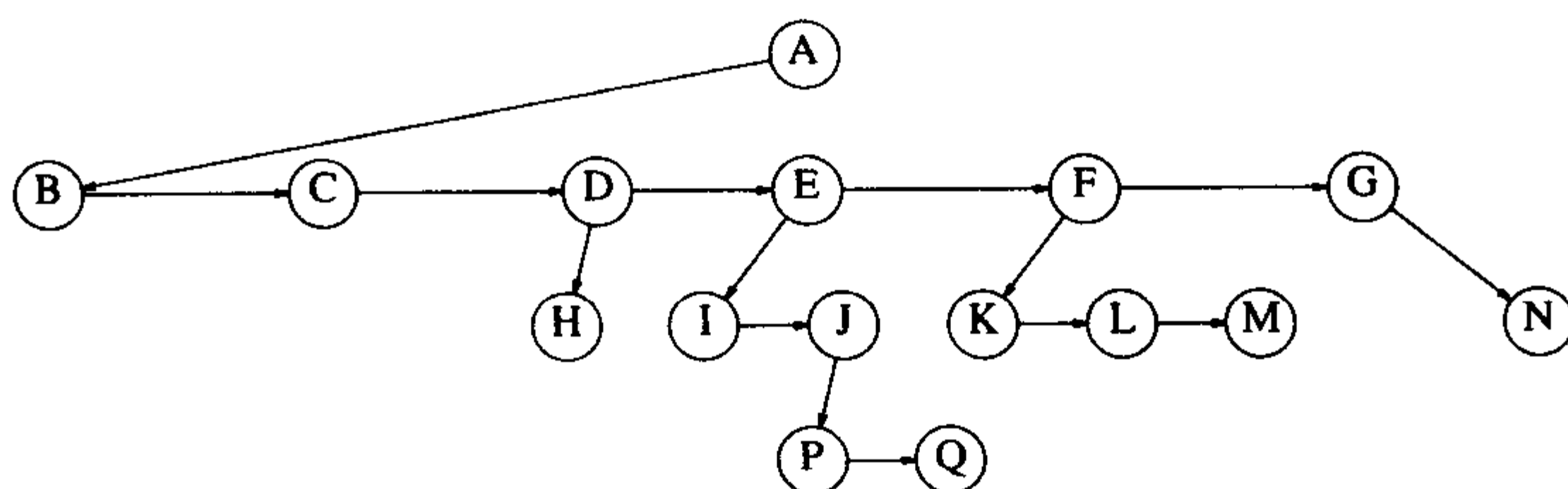


图4-4 在图4-2中所表示的树的第一儿子/下一兄弟的表示法

### 4.1.2 树的遍历及应用

树有很多应用。流行的用法之一是用于包括UNIX和DOS在内的许多常用操作系统中的目录结构。图4-5是UNIX文件系统中的—个典型的目录。

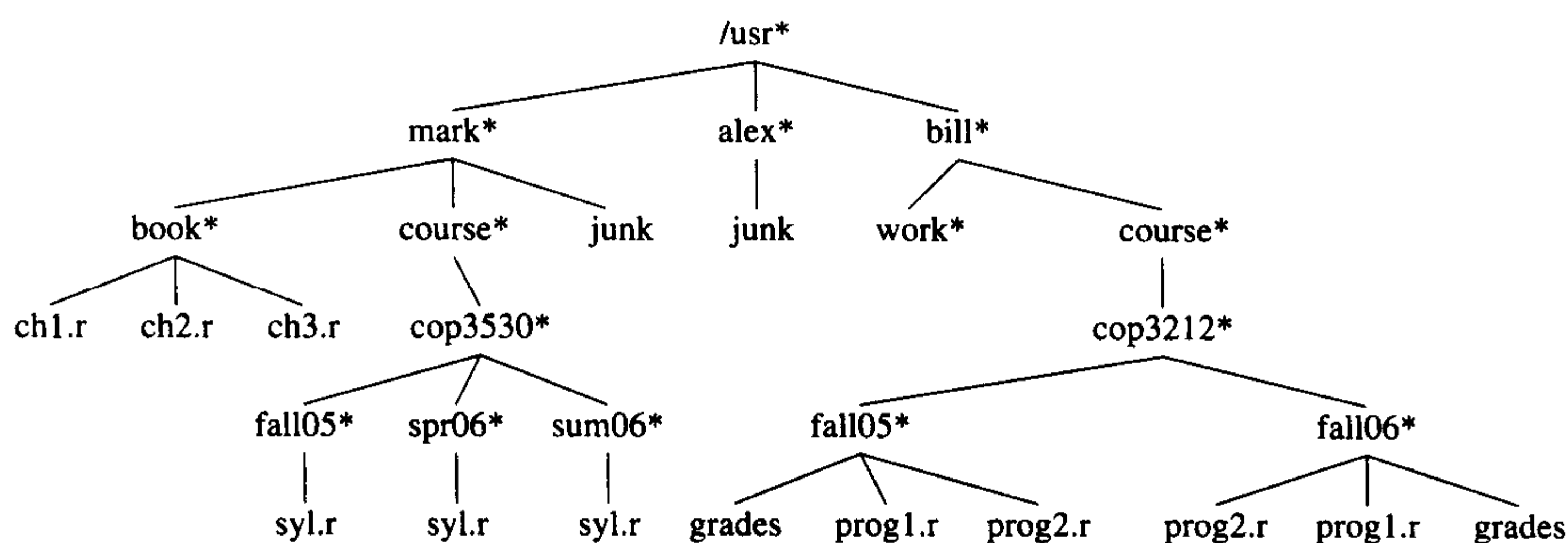


图4-5 UNIX目录

这个目录的根是`/usr`（名字后面的星号指出`/usr`本身就是一个目录）。`/usr`有三个儿子：`mark`、`alex`和`bill`，它们自己也都是目录。因此，`/usr`包含三个目录并且没有常规的文件。文件名`/usr/mark/book/ch1.r`先后三次通过最左边的儿子结点而得到。第一个“/”后的每个“/”都表示一条边；结果为一全路径名（pathname）。这个分级文件系统非常流行，因为它使得用户能够逻辑地组织数据。不仅如此，在不同目录下的两个文件还可以享有相同的名字，因为它们必然有从根开始的不同的路径从而具有不同的路径名。在UNIX文件系统中的目录就是含有它的所有儿子的一个文件，因此，这些目录几乎是完全按照上述的类型声明构造的<sup>1</sup>。事实上，按照UNIX的某些版本，如果将打印文件的标准命令应用到目录上，那么目录中的文件名能够在（与其他非ASCII信息一起的）输出中看到。

设我们想要列出目录中所有文件的名字。输出格式将是：深度为 $d_i$ 的文件将被 $d_i$ 次跳格（tab）缩进后打印其名。该算法在图4-6中以伪码给出。

```
void FileSystem::listAll( int depth = 0 ) const
{
1  printName( depth ); // Print the name of the object
2  if( isDirectory( ) )
3      for each file c in this directory (for each child)
4          c.listAll( depth + 1 );
}
```

图4-6 列出分级文件系统中目录的伪码

1. 在 UNIX 文件系统中每个目录还有一项指向该目录本身以及另一项指向该目录的父目录。因此，严格说来，UNIX 文件系统不是树，而是类树（treelike）。

为了显示根时不进行缩进，递归函数listAll需要从深度0开始。这里的深度是一个内部簿记变量，而不是主调例程能够期望知道的那种参数。因此，需要给depth提供默认值0。

算法的逻辑简单易懂。文件对象的名字以适当个数的跳格打印出来。如果是一个目录，那么我们递归地一个一个地处理它所有的儿子。这些儿子处在同一个深度上，因此需要缩进一个附加的空间。整个输出如图4-7中所示。

116

```

/usr
  mark
    book
      ch1.r
      ch2.r
      ch3.r
    course
      cop3530
        fall05
          syl.r
        spr06
          syl.r
        sum06
          syl.r
      junk
    alex
      junk
    bill
      work
      course
        cop3212
          fall05
            grades
            prog1.r
            prog2.r
          fall06
            prog2.r
            prog1.r
            grades

```

图4-7 （前序）目录列表

这种遍历的策略称为**前序遍历**（preorder traversal）。在前序遍历中，对结点的处理工作是在它的诸儿子结点被处理之前进行的。当该程序运行时，显然第1行对每个结点恰好执行一次，因为每个名字只输出一次。由于第1行对每个结点最多执行一次，因此第2行也必然对每个结点执行一次。不仅如此，对于每个结点的每一个儿子结点第4行最多只能被执行一次。不过，儿子的个数恰好比结点的个数少1。最后，第4行每执行一次，for循环就迭代一次，每当循环结束时再加上一次。因此，每个结点总的工作量是常数。如果有 $N$ 个文件名需要输出，则运行时间就是 $O(N)$ 。

117

另一种遍历树的常用方法是**后序遍历**（postorder traversal）。在后序遍历中，在一个结点的工作是在它的诸儿子结点被计算后进行的。例如，图4-8表示的是与前面相同的目录结构，其中圆括号内的数代表每个文件占用的磁盘块的个数。

由于目录本身也是文件，因此它们也有大小。设我们想要计算被该树所有文件占用的磁盘块的总数。最常见的做法是找出含于子目录/usr/mark（30）、/usr/alex（9）和/usr/bill（32）的块的个数。于是，磁盘块的总数就是子目录中的块的总数（71）加上/usr使用的一个块，共72个块。图4-9中的伪码方法size实现了这种遍历策略。



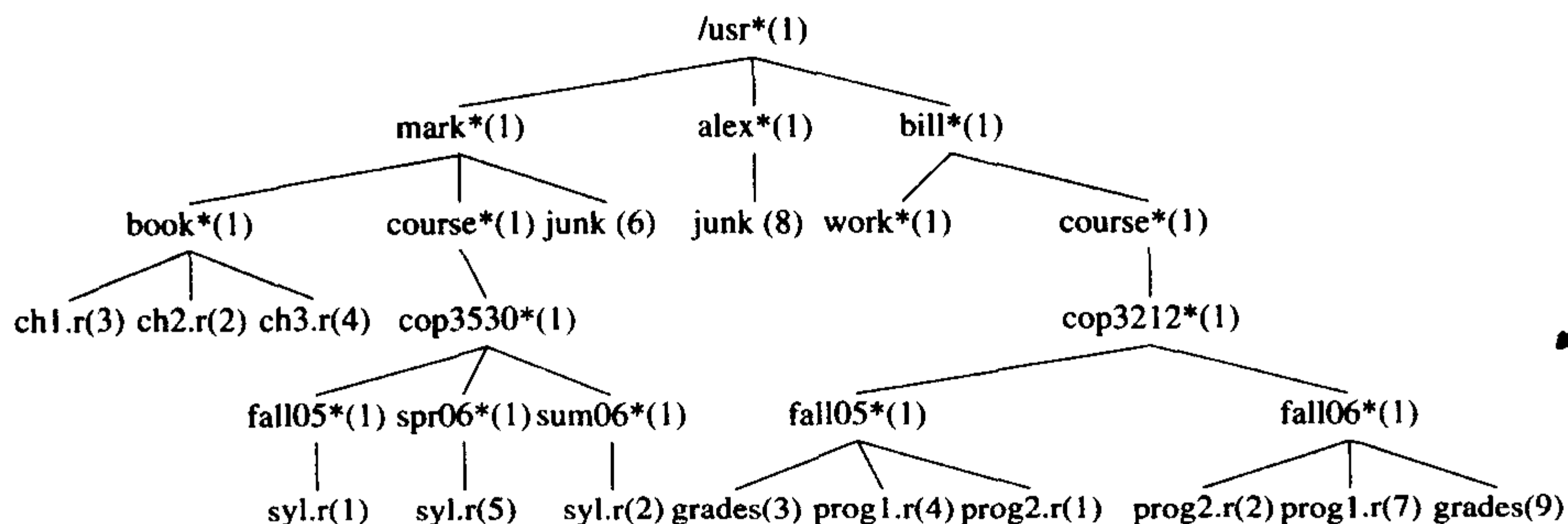


图4-8 经由后序遍历得到的带有文件大小的UNIX目录

```

int FileSystem::size( ) const
{
    int totalSize = sizeofThisFile( );

    if( isDirectory( ) )
        for each file c in this directory (for each child)
            totalSize += c.size( );

    return totalSize;
}

```

图4-9 计算一个目录大小的伪码例程

如果当前对象不是一个目录，那么size只返回它所占用的块数。否则，被该目录占用的块数将被加到其所有子结点（递归地）找到的块数中去。为了区别后序遍历策略和前序遍历策略，图4-10显示了每个目录或文件的大小是如何由该算法产生的。

118

|         |    |
|---------|----|
| chl.r   | 3  |
| ch2.r   | 2  |
| ch3.r   | 4  |
| book    | 10 |
| syl.r   | 1  |
| fall05  | 2  |
| syl.r   | 5  |
| spr06   | 6  |
| syl.r   | 2  |
| sum06   | 3  |
| cop3530 | 12 |
| course  | 13 |
| junk    | 6  |
| mark    | 30 |
| junk    | 8  |
| alex    | 9  |
| work    | 1  |
| grades  | 3  |
| prog1.r | 4  |
| prog2.r | 1  |
| fall05  | 9  |
| prog2.r | 2  |
| prog1.r | 7  |
| grades  | 9  |
| fall06  | 19 |
| cop3212 | 29 |
| course  | 30 |
| bill    | 32 |
| /usr    | 72 |

图4-10 函数size的踪迹

4.2 二叉树

二叉树（binary tree）是一棵每个结点都不能有多于两个儿子的树。  
图4-11显示一棵由一个根和两棵子树组成的二叉树，子树 $T_L$ 和 $T_R$ 均可能为空。

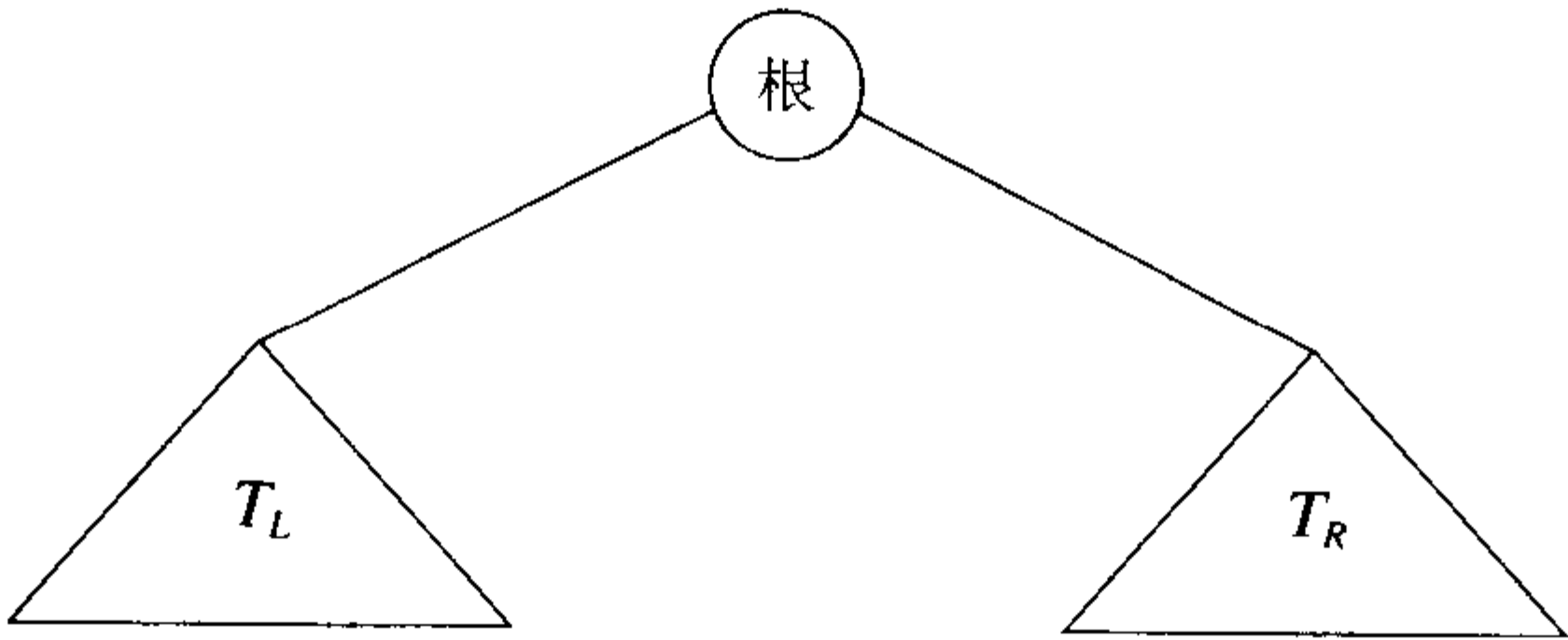


图4-11 一般二叉树

二叉树的一个性质是平均二叉树的深度要比结点个数 $N$ 小得多，这个性质有时很重要。分析表明，这个平均深度为 $O(\sqrt{N})$ ，而对于特殊类型的二叉树，即二叉查找树（binary search tree），其深度的平均值是 $O(\log N)$ 。遗憾的是，正如图4-12中的例子所示，这个深度也可以大到 $N-1$ 。

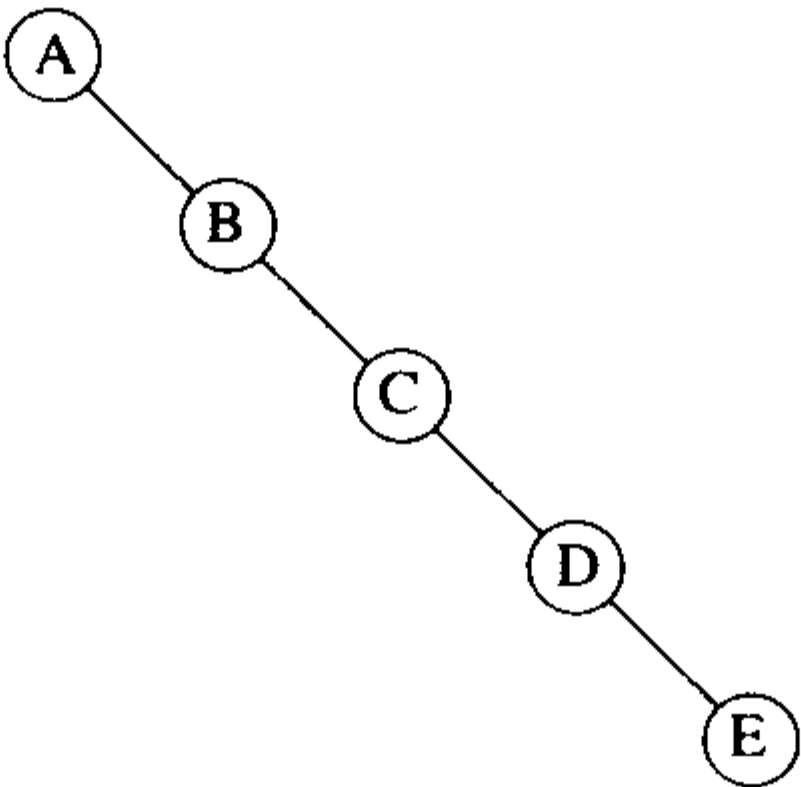


图4-12 最坏情形的二叉树

4.2.1 实现

因为一个二叉树结点最多有两个儿子，所以可以直接链接到它们。树结点的声明在结构上类似于双向链表的声明。在声明中，一个结点就是由element（元素）的信息加上两个到其他结点的引用（left和right）组成的结构（见图4-13）。

```
struct BinaryNode
{
    Object      element;          // The data in the node
    BinaryNode *left;             // Left child
    BinaryNode *right;            // Right child
};
```

图4-13 二叉树结点类（伪代码）

可以用在画链表时惯用的矩形框画出二叉树，但是，树一般画成用一些直线连接起来的圆圈，因为二叉树实际上就是图（graph）。当涉及树时，我们也不显式地画出NULL链，因为具有 $N$ 个结点的每一棵二叉树都将需要 $N+1$ 个NULL链。

二叉树有许多与搜索无关的重要应用。二叉树的主要用处之一是在编译器的设计领域，下面就来探讨这个问题。

### 4.2.2 一个例子——表达式树

图4-14是一个表达式树 (expression tree) 的例子。表达式树的树叶是操作数 (operand)，如常数或变量名字，而其他的结点为操作符 (operator)。由于这里所有的操作都是二元的，因此这棵特定的树正好是二叉树，虽然这是最简单的情况，但是结点还是有可能含有多于两个的儿子。一个结点也有可能只有一个儿子，如具有一元减运算符 (unary minus operator) 的情形。可以将通过递归计算左子树和右子树所得到的值应用在根处的操作符上而算出表达式树  $T$  的值。在我们的例子中，左子树的值是  $a + (b * c)$ ，右子树的值是  $((d * e) + f) * g$ ，因此整个树表示  $(a + (b * c)) + ((d * e) + f) * g$ 。

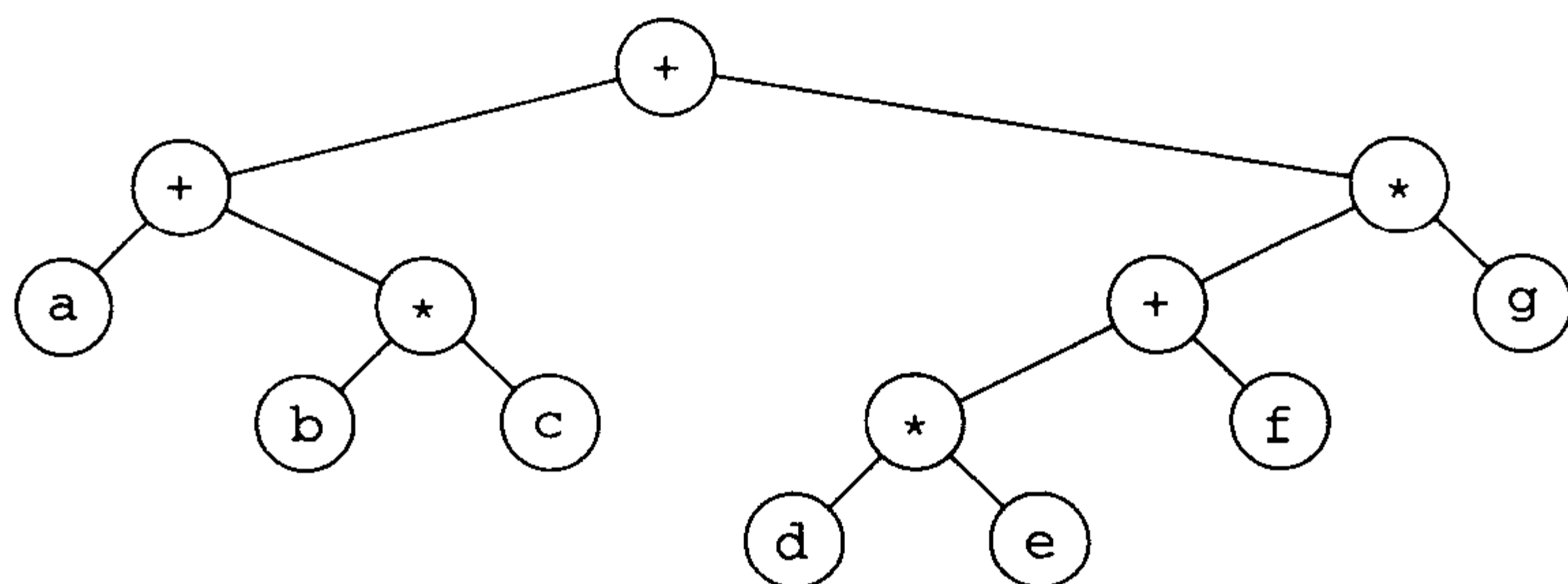


图4-14  $(a+b*c) + ((d*e+f)*g)$  的表达式树

可以通过递归地产生一个带括号的左表达式，然后打印出在根处的操作符，最后再递归地产生一个带括号的右表达式而得到一个 (对两个括号整体进行运算的) 中缀表达式 (infix expression)。这种一般的方法 (左，结点，右) 称为中序遍历 (inorder traversal)；由于其产生的表达式类型，这种遍历很容易记忆。

另一个遍历策略是递归地打印出左子树、右子树，然后打印操作符。如果应用这种策略于上面的树，则输出将是  $a\ b\ c\ *\ +\ d\ e\ *\ f\ +\ g\ *\ +$ ，容易看出，这就是3.6.3节中的后缀表示法。这种遍历策略一般称为后序遍历 (postorder traversal)。我们已在4.1节见过这种遍历策略。

第三种遍历策略是先打印出操作符，然后递归地打印出左子树和右子树。其结果  $++a*bc*++*defg$ ，这是不太常用的前缀 (prefix) 记法，这种遍历策略称为前序遍历 (preorder traversal)，也已在4.1节见过它。本章后面还要讨论这些遍历方法。

121

#### 构造一棵表达式树

下面给出一种算法来把后缀表达式转变成表达式树。由于我们已经有了将中缀表达式转变成后缀表达式的算法，因此可以从这两种常用类型的输入生成表达式树。所描述的方法酷似3.6.3节的后缀求值算法。我们一次一个符号地读入表达式。如果符号是操作数，那么就建立一个单结点树并将它推入栈中。如果符号是操作符，那么就从栈中弹出两棵树  $T_1$  和  $T_2$  ( $T_1$  先弹出) 并形成一棵新的树，该树的根就是操作符，它的左、右儿子分别是  $T_2$  和  $T_1$ 。然后将指向这棵新树的指针压入栈中。

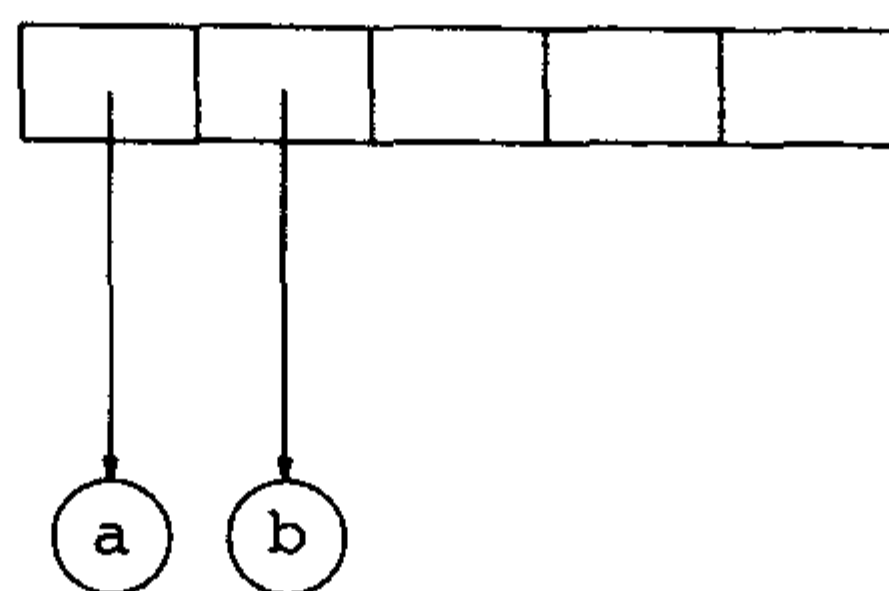
下面来看一个例子。设输入为

$a\ b\ +\ c\ d\ e\ +\ *\ +$

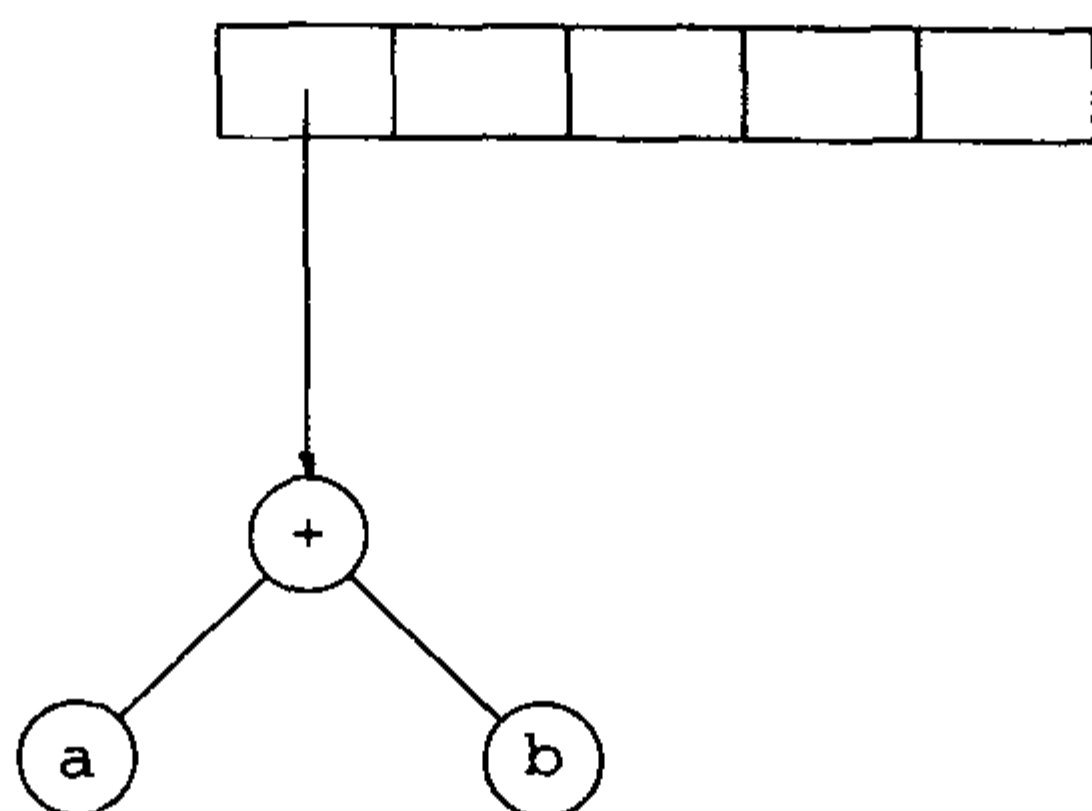
前两个符号是操作数，因此创建两棵单结点树并将指向它们的指针压入栈中。<sup>1</sup>

1. 为了方便起见，我们将让图中的栈从左到右增长。

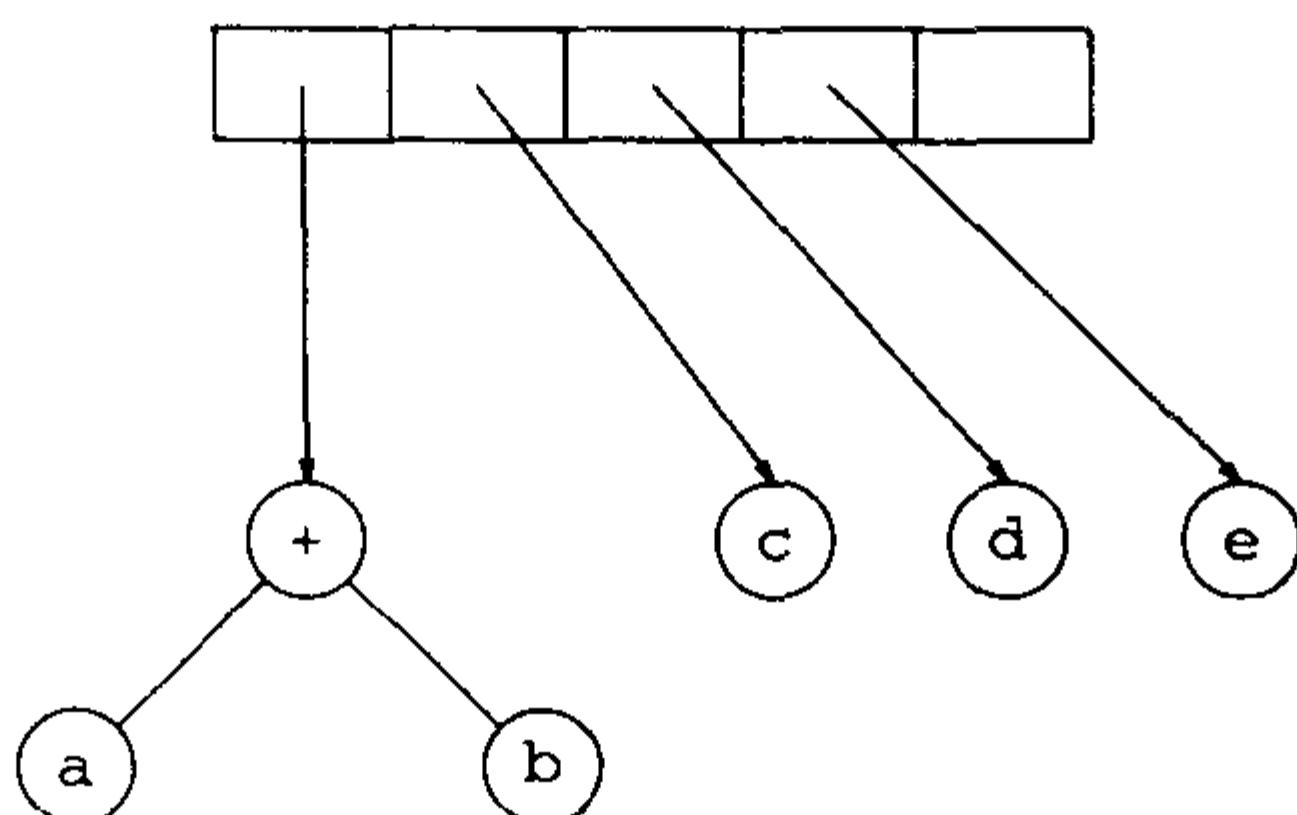




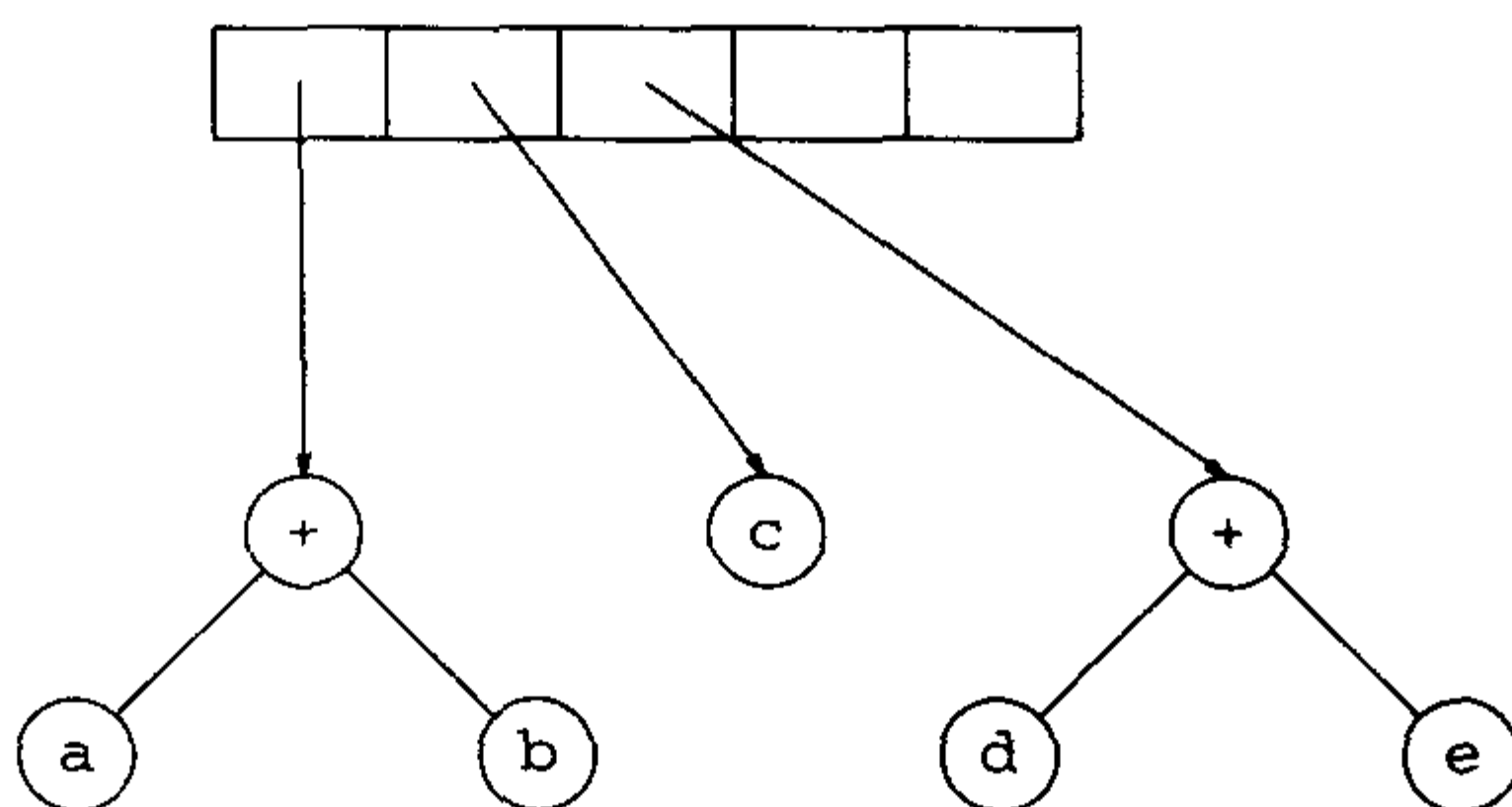
接着，“+”被读入，因此指向两棵树的指针被弹出，形成一棵新的树，并将指向它的指针压入栈中。



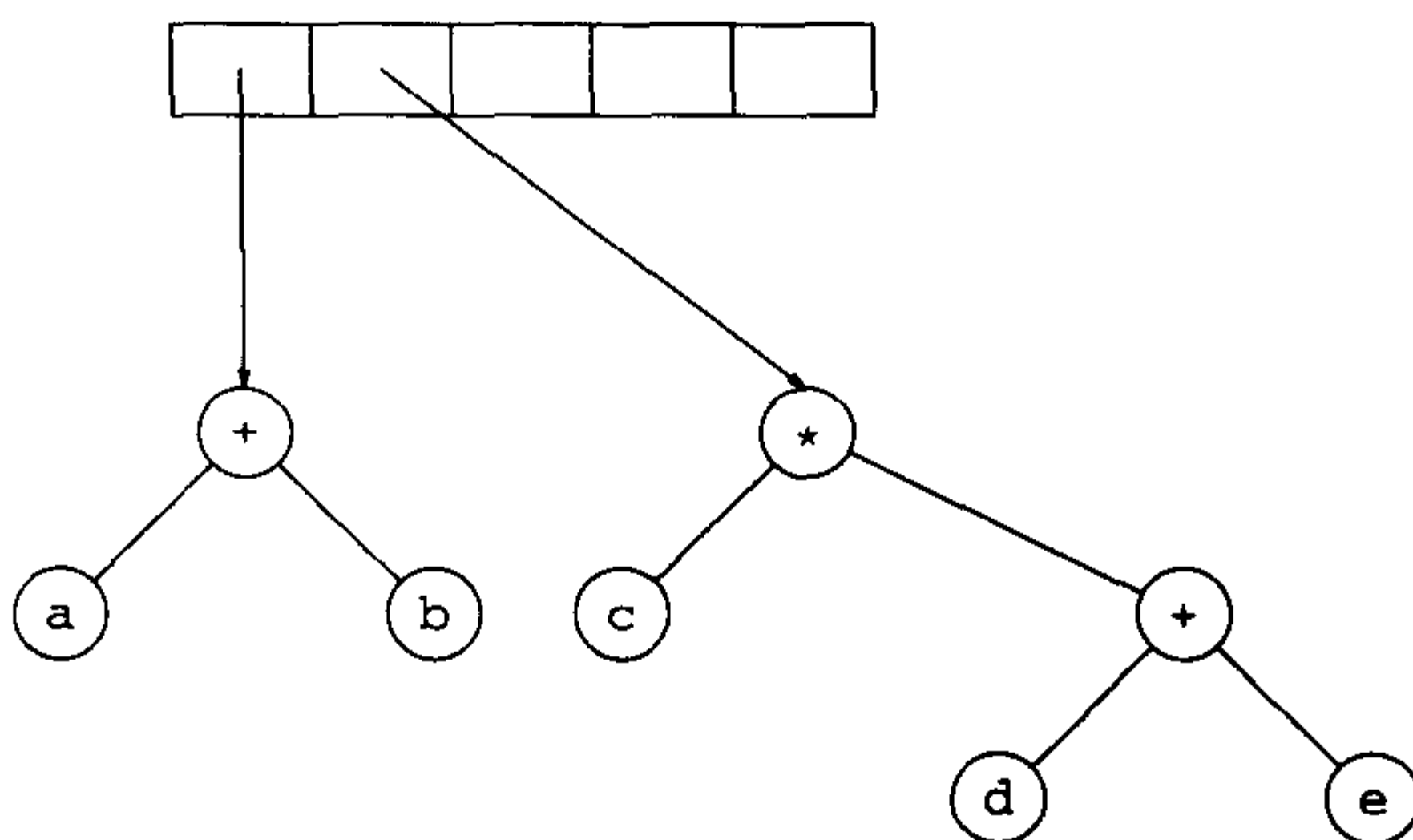
122 然后，c、d和e被读入，在每个单结点树创建后，指向对应的树的指针被压入栈中。



接下来读入“+”号，因此两棵树合并。

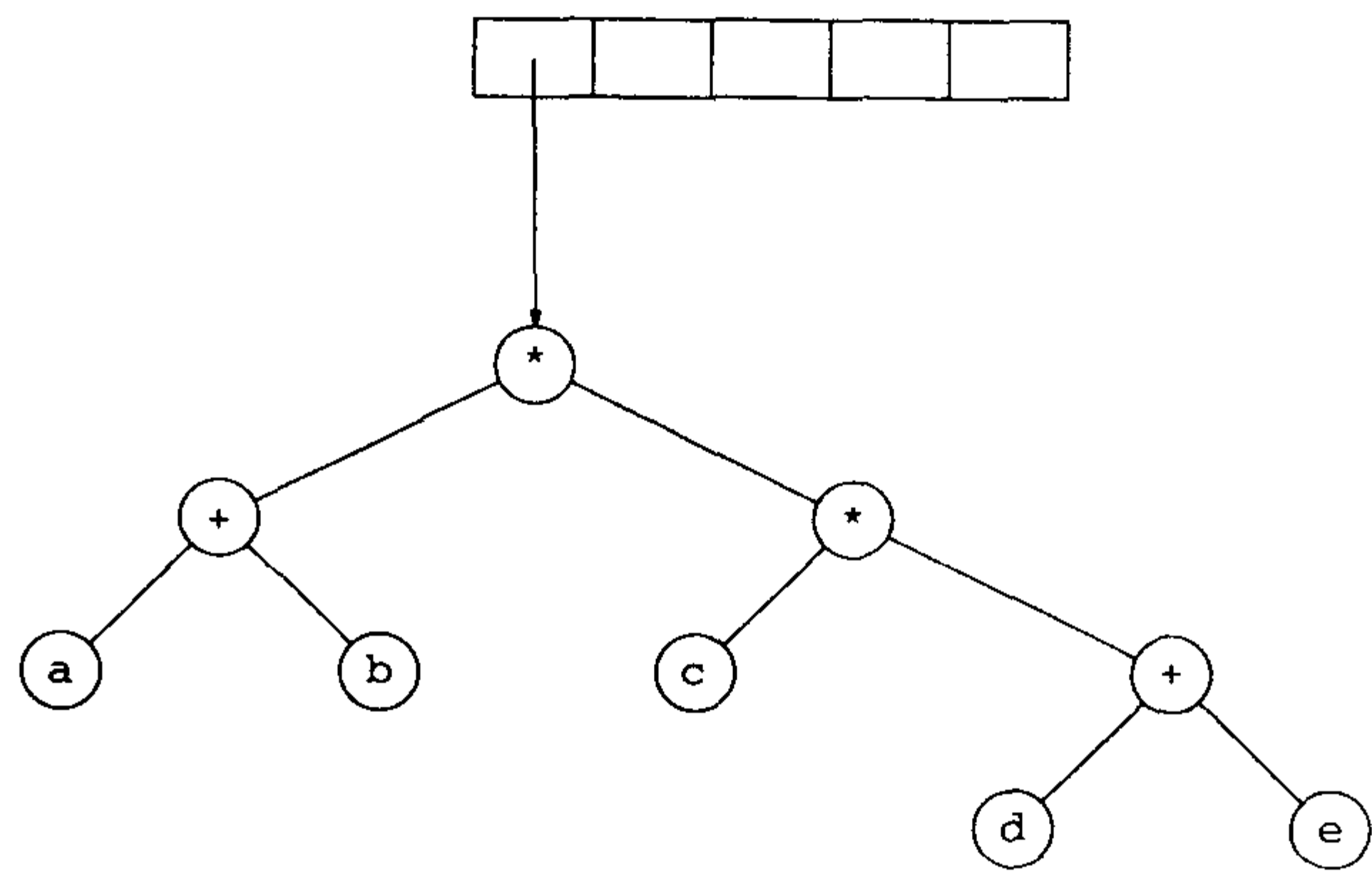


继续进行，读入“\*”号，因此，弹出两棵树的指针并形成一棵新的树，“\*”号是它的根。



最后，读入最后一个符号，两棵树合并，而指向最后的树的指针被留在栈中。

123



### 4.3 查找树ADT——二叉查找树

二叉树的一个重要的应用是它们在查找中的应用。假设树中的每个结点存储一项数据。在我们的例子中，虽然任意复杂的项在C++中都很容易处理，但为简单起见还是假设它们是些整数。我们还将假设，所有的项都是互异的，以后再处理有重复元的情况。

使二叉树成为二叉查找树的性质是，对于树中的每个结点 $X$ ，它的左子树中所有项的值小于 $X$ 中的项，而它的右子树中所有项的值大于 $X$ 中的项。注意，这意味着，该树所有的元素都可以用某种一致的方式排序。在图4-15中，左边的树是二叉查找树，但右边的树却不是。右边的树在其项是6的结点（该结点正好是根结点）的左子树中，有一个结点的项是7。

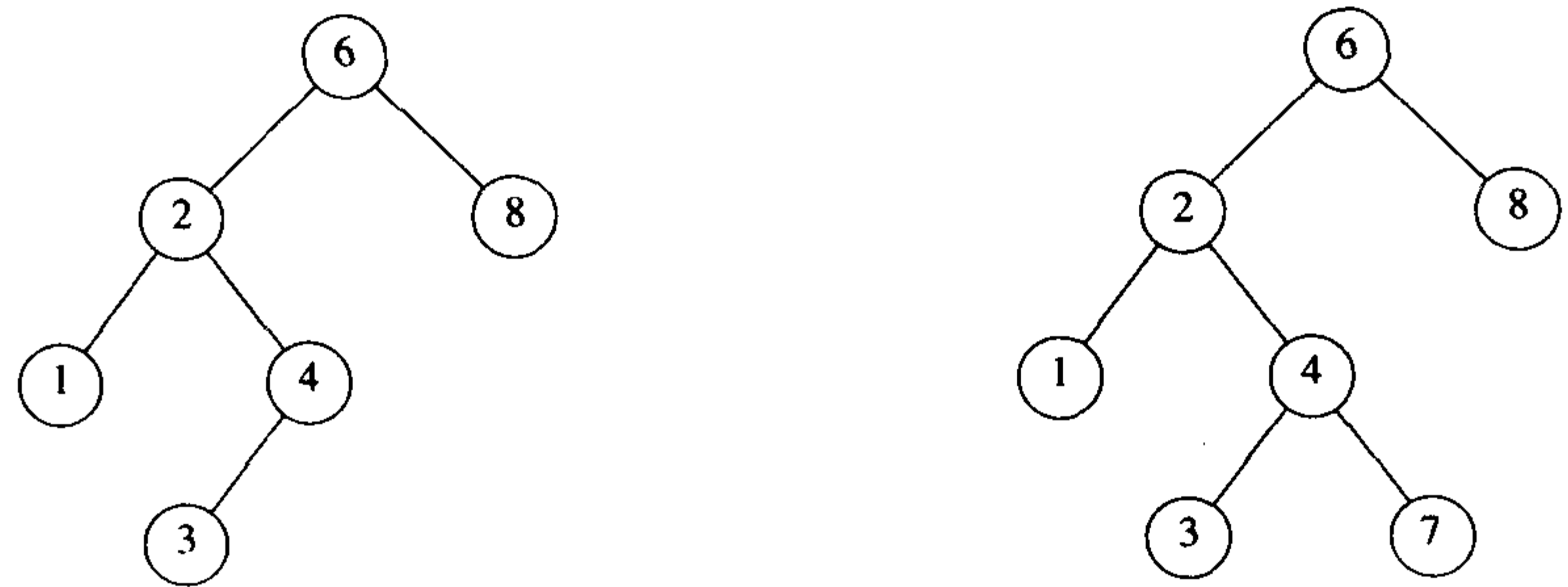


图4-15 两棵二叉树（只有左边的树是查找树）

现在给出通常对二叉查找树进行的操作的简要描述。注意，由于树的递归定义，通常是递归地编写这些操作的例程。因为二叉查找树的平均深度是 $O(\log N)$ ，所以我们一般不必担心栈空间被用尽。

图4-16所示是BinarySearchTree类模板的接口。这里有件事值得注意。查找是基于“<”操作符的，而该操作符对特定的Comparable类型必须进行定义。特别是，若 $x < y$ 和 $y < x$ 均为false，那么，项 $x$ 与项 $y$ 匹配。这允许Comparable是一个复杂类型（例如雇员记录），而该复杂类型中只有一部分类型（例如社保号数据成员或工资）具有比较功能。1.6.3节例举了设计一个可以作为Comparable使用的类的一般技术。在4.3.1节中的一个可选方案则适用于函数对象。

```

1  template <typename Comparable>
2  class BinarySearchTree
3  {
4      public:
5          BinarySearchTree( );
6          BinarySearchTree( const BinarySearchTree & rhs );
7          ~BinarySearchTree( );
8
9          const Comparable & findMin( ) const;
10         const Comparable & findMax( ) const;
11         bool contains( const Comparable & x ) const;
12         bool isEmpty( ) const;
13         void printTree( ) const;
14
15         void makeEmpty( );
16         void insert( const Comparable & x );
17         void remove( const Comparable & x );
18
19         const BinarySearchTree & operator=( const BinarySearchTree & rhs );
20
21     private:
22         struct BinaryNode
23         {
24             Comparable element;
25             BinaryNode *left;
26             BinaryNode *right;
27
28             BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
29                 : element( theElement ), left( lt ), right( rt ) { }
30         };
31
32         BinaryNode *root;
33
34         void insert( const Comparable & x, BinaryNode * & t ) const;
35         void remove( const Comparable & x, BinaryNode * & t ) const;
36         BinaryNode * findMin( BinaryNode *t ) const;
37         BinaryNode * findMax( BinaryNode *t ) const;
38         bool contains( const Comparable & x, BinaryNode *t ) const;
39         void makeEmpty( BinaryNode * & t );
40         void printTree( BinaryNode *t ) const;
41         BinaryNode * clone( BinaryNode *t ) const;
42 };

```

图4-16 二叉查找树类的框架

数据成员是指向树根结点的指针，该指针对空树为NULL。public的成员函数使用调用private递归函数的常规技术。图4-17是对contains、insert和remove的这种调用方式的例子。

几个private成员函数使用引址调用来传递指针变量的技术。这允许public成员函数将指向树根的指针传递给private递归成员函数。然后递归函数就可以改变根的值，于是root就可以指向其他的结点。我们在研究insert代码的时候将讨论该技术的更多细节。

现在我们可以描述某些private方法。

### 4.3.1 contains

如果在树 $T$ 中有项为 $X$ 的结点，那么contains操作就返回true，否则，若没有这样的结点就返回false。树结构使得该操作很简单。如果 $T$ 为空，那么就可以返回false；否则，如果存

储在 $T$ 中的项为 $X$ ，就返回true。若以上两种情况都不成立，就对 $T$ 的一个子树进行递归调用。至于是左子树还是右子树取决于 $X$ 与存储在 $T$ 中的项的关系。图4-18所示的代码是这种策略的实现。

```

1  /**
2   * Returns true if x is found in the tree.
3   */
4  bool contains( const Comparable & x ) const
5  {
6      return contains( x, root );
7  }
8
9  /**
10   * Insert x into the tree; duplicates are ignored.
11   */
12 void insert( const Comparable & x )
13 {
14     insert( x, root );
15 }
16
17 /**
18   * Remove x from the tree. Nothing is done if x is not found.
19   */
20 void remove( const Comparable & x )
21 {
22     remove( x, root );
23 }

```

图4-17 公有成员函数调用私有递归成员函数的示例

```

1  /**
2   * Internal method to test if an item is in a subtree.
3   * x is item to search for.
4   * t is the node that roots the subtree.
5   */
6  bool contains( const Comparable & x, BinaryNode *t ) const
7  {
8      if( t == NULL )
9          return false;
10     else if( x < t->element )
11         return contains( x, t->left );
12     else if( t->element < x )
13         return contains( x, t->right );
14     else
15         return true;    // Match
16 }

```

图4-18 二叉查找树的contains操作

注意测试的顺序。关键的问题是首先要对是否为空树进行测试，因为如果不这么做就会产生一个企图通过NULL指针访问数据成员的运行错误。其余的测试应该使得最不可能的情况安排在最后进行。还要注意，这里的两个递归调用事实上都是尾递归并且可以用一个while循环很容易地代替。尾递归的使用在这里是合理的，因为算法表达式的简明性是以速度的降低为代价的，而这里所使用的栈空间的量也只不过是 $O(\log N)$ 而已。

图4-19给出了使用函数对象而不是使用Comparable项所需要做的微小修改。这模拟了1.6节的常用例程。



```

1  template <typename Object, typename Comparator=less<Object> >
2  class BinarySearchTree
3  {
4      public:
5
6          // Same methods, with Object replacing Comparable
7
8      private:
9
10         BinaryNode *root;
11         Comparator isLessThan;
12
13         // Same methods, with Object replacing Comparable
14
15         /**
16          * Internal method to test if an item is in a subtree.
17          * x is item to search for.
18          * t is the node that roots the subtree.
19          */
20         bool contains( const Object & x, BinaryNode *t ) const
21         {
22             if( t == NULL )
23                 return false;
24             else if( isLessThan( x, t->element ) )
25                 return contains( x, t->left );
26             else if( isLessThan( t->element, x ) )
27                 return contains( x, t->right );
28             else
29                 return true;    // Match
30         }
31 };

```

图4-19 使用函数对象实现二叉查找树的示例

### 4.3.2 findMin和findMax

这两个private例程分别返回指向树中包含最小元和最大元的结点的指针。为执行findMin，从根开始并且只要有左儿子就向左进行。终止点就是最小的元素。findMax例程除分支朝向右儿子外其余过程相同。

这种递归是如此容易，以至于许多程序设计员不厌其烦地使用它。我们用两种方法编写两个例程，用递归编写findMin而用非递归编写findMax（见图4-20和图4-21）。

```

1  /**
2   * Internal method to find the smallest item in a subtree t.
3   * Return node containing the smallest item.
4   */
5  BinaryNode * findMin( BinaryNode *t ) const
6  {
7      if( t == NULL )
8          return NULL;
9      if( t->left == NULL )
10         return t;
11     return findMin( t->left );
12 }

```

图4-20 对二叉查找树的findMin的递归实现

```

1  /**
2   * Internal method to find the largest item in a subtree t.
3   * Return node containing the largest item.
4   */
5  BinaryNode * findMax( BinaryNode *t ) const
6  {
7      if( t != NULL )
8          while( t->right != NULL )
9              t = t->right;
10     return t;
11 }

```

图4-21 对二叉查找树的findMax的非递归实现

注意我们是如何小心地处理空树的退化情况的。虽然这么做总是重要的，但在递归程序中这么做尤其重要。此外，还要注意，在findMax中对t的改变是安全的，因为我们只用指针的副本来进行工作。不管怎么说，还是应该随时特别小心，因为诸如t->right=t->right->right这样的语句将会产生一些变化。

128

### 4.3.3 insert

进行插入操作的例程在概念上是很简单的。为了将X插入到树T中，可以像使用contains那样沿着树查找。如果找到X，则什么也不用做（或做一些“更新”）。否则，将X插入到遍历的路径上的最后一点上。图4-22显示了实际的插入情况。为了插入5，我们遍历该树，就好像在运行contains。在具有项4的结点处，我们需要向右行进，但右边不存在子树，因此5不在这棵树上，从而这个位置就是所要插入的位置。

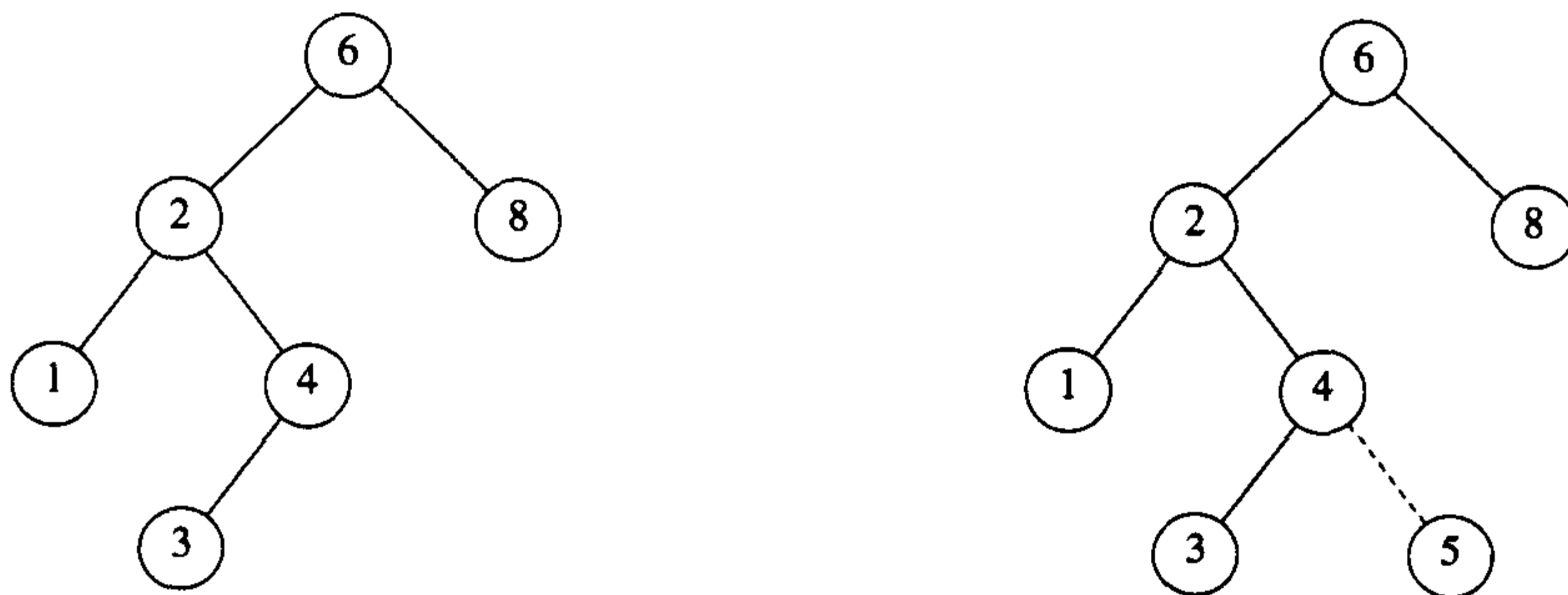


图4-22 在插入5以前和以后的二叉查找树

重复元的插入可以通过在结点记录中保留一个附加字段以指示此数据元出现的频率来处理。这使整个树增加了某些附加空间，但是，却比将重复信息放到树中要好（它将使树的深度变得很大）。当然，如果“<”操作符使用的键只是一个更大的结构的一部分，那么这种方法行不通。此时可以把具有相同键的所有结构保留在一个辅助数据结构中，如表或是另一棵查找树。

图4-23给出了插入例程的代码。第12行和第14行递归地将x插入到适当的子树中。注意，在递归例程中，只有当一个新的树叶生成的时候，t才改变。当这种情况发生的时候，就说明递归例程被其他结点p调用了。该结点p是树叶的父亲。调用将是insert(x,p->left)或者insert(x,p->right)。在任何一种方法中，t是到p->left或p->right的引用。这意味着p->left或p->right将会被改变为指向新结点。总而言之，这是一个老套的技巧。

129

```

1  /**
2  * Internal method to insert into a subtree.
3  * x is the item to insert.
4  * t is the node that roots the subtree.
5  * Set the new root of the subtree.
6  */
7  void insert( const Comparable & x, BinaryNode * & t )
8  {
9      if( t == NULL )
10         t = new BinaryNode( x, NULL, NULL );
11     else if( x < t->element )
12         insert( x, t->left );
13     else if( t->element < x )
14         insert( x, t->right );
15     else
16         ; // Duplicate; do nothing
17 }

```

图4-23 将元素插入到二叉查找树的例程

#### 4.3.4 remove

同许多数据结构一样，最困难的操作是删除。一旦发现要被删除的结点，就需要考虑几种可能的情况。

如果结点是一片树叶，那么它可以被立即删除。如果结点有一个儿子，则该结点可以在其父结点调整它的链以绕过该结点后被删除（为了清楚起见，我们将明确地画出链的指向），见图4-24。

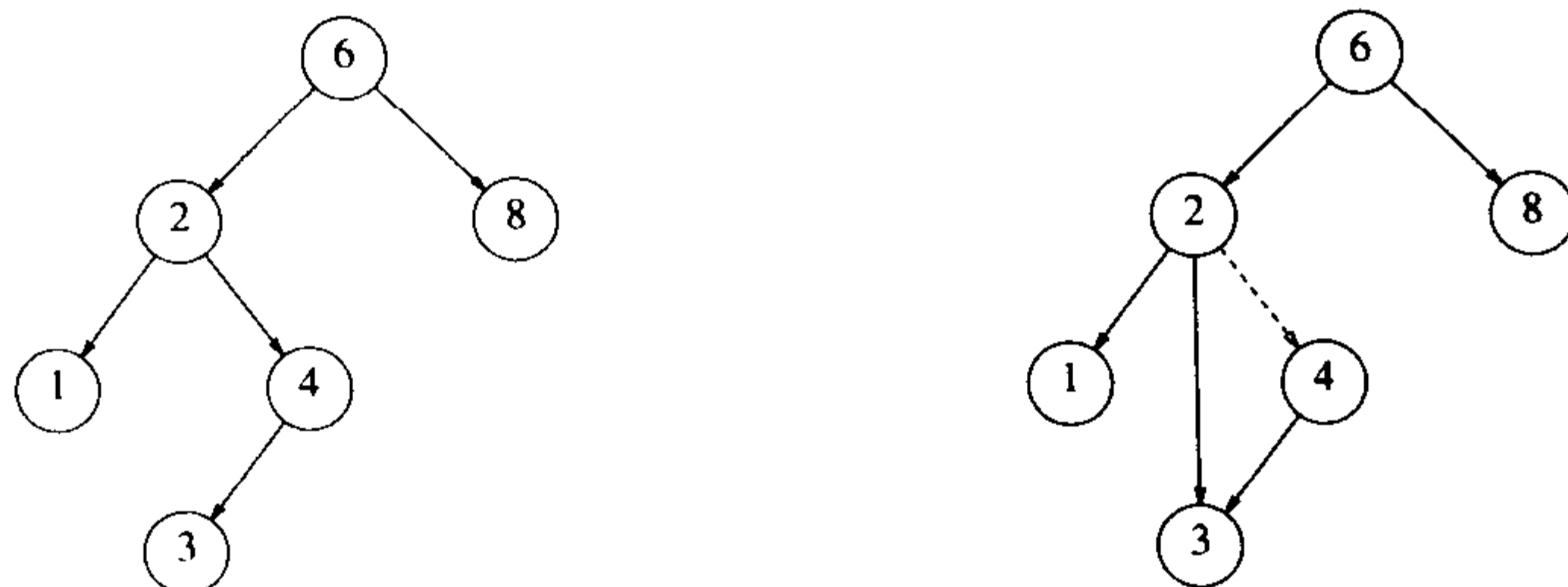


图4-24 具有一个儿子的结点4删除前后的情况

复杂的情况是处理具有两个儿子的结点。一般的删除策略是用其右子树的最小的数据（很容易找到）代替该结点的数据并递归地删除那个结点（现在它是空的）。因为右子树中的最小的结点不可能有左儿子，所以第二次remove就很容易。图4-25显示了一棵初始的树及其中一个结点被删除后的结果。要被删除的结点是根的左儿子；其键值是2。它被右子树中的最小数据3所代替，然后原结点如前例那样被删除。

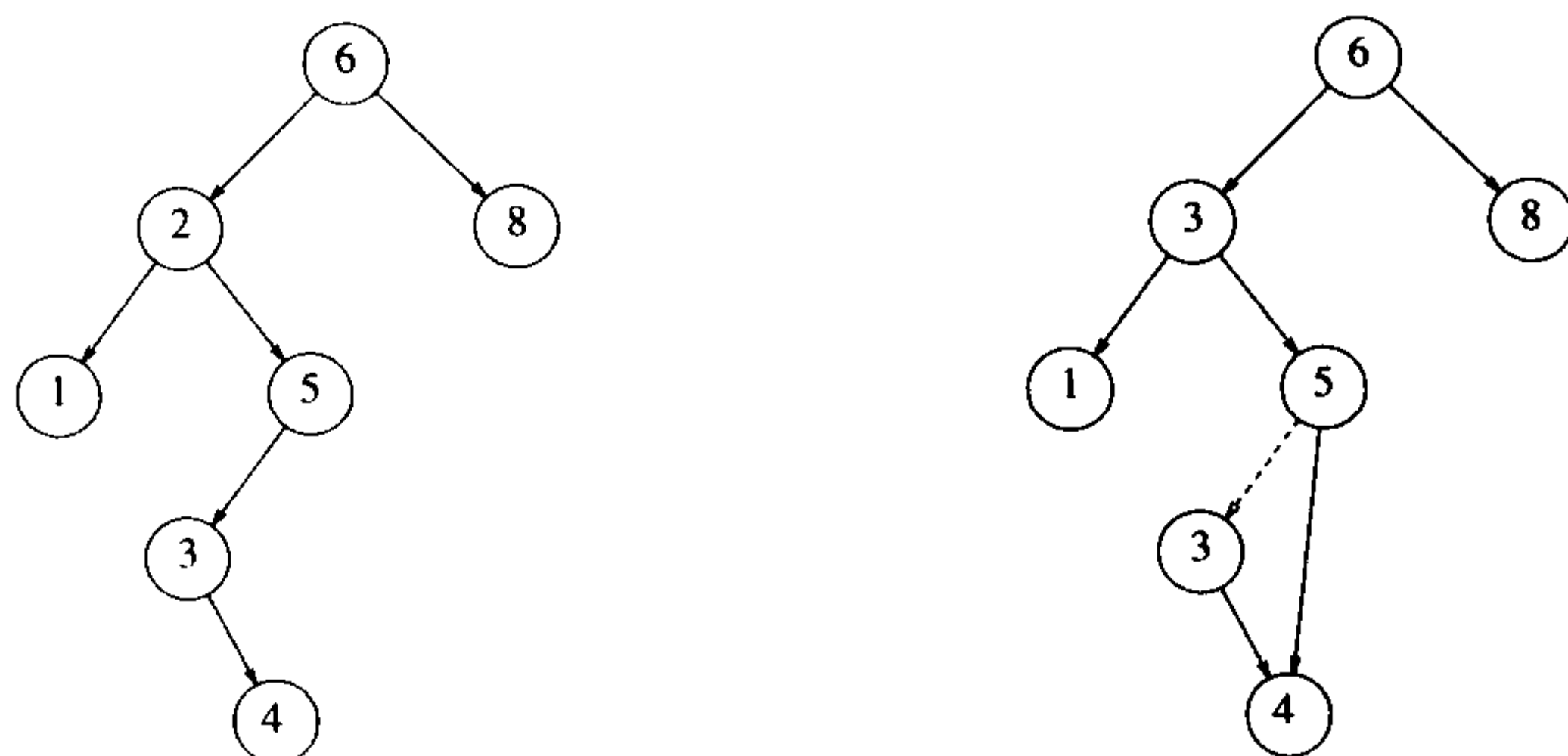


图4-25 删除具有两个儿子的结点2前后的情况

图4-26中的程序可以完成删除的工作，但其效率并不高，因为它沿该树进行两次搜索以查找和删除右子树中最小的结点。通过写一个特殊的removeMin方法可以容易地改变效率不高的缺点，这里将其略去只是为了简明紧凑。

131

```

1  /**
2   * Internal method to remove from a subtree.
3   * x is the item to remove.
4   * t is the node that roots the subtree.
5   * Set the new root of the subtree.
6   */
7  void remove( const Comparable & x, BinaryNode * & t )
8  {
9      if( t == NULL )
10         return;    // Item not found; do nothing
11     if( x < t->element )
12         remove( x, t->left );
13     else if( t->element < x )
14         remove( x, t->right );
15     else if( t->left != NULL && t->right != NULL ) // Two children
16     {
17         t->element = findMin( t->right )->element;
18         remove( t->element, t->right );
19     }
20     else
21     {
22         BinaryNode *oldNode = t;
23         t = ( t->left != NULL ) ? t->left : t->right;
24         delete oldNode;
25     }
26 }

```

图4-26 二叉查找树的删除例程

如果删除的次数不多，则通常使用的策略是**懒惰删除**（lazy deletion）：当一个元素要被删除时，它仍留在树中，而只是做了个被删除的记号。这种做法在有重复项时很流行，因为此时记录出现频率数的数据成员可以减1。如果树中的实际结点数和“被删除”的结点数相同，那么树的深度预计只上升一个小的常数（为什么？），因此，只存在一个与懒惰删除相关的非常小的时间损耗。再有，如果被删除的项又被重新插入，那么分配一个新单元的开销就避免了。

#### 4.3.5 析构函数和复制赋值操作符

与往常一样，析构函数调用makeEmpty。公有的makeEmpty（没有显示出来）则简单地调用私有的递归版本的makeEmpty。如图4-27所示，在递归地处理t的子树之后，对t调用delete。这样一来，所有的结点就都递归地回收了。注意，在最后，t（此时为root）就改为指向NULL。在图4-28中显示的复制赋值操作符与以往的程序一样，首先调用makeEmpty来回收内存，然后进行rhs的复制。我们使用一个名为clone的很老套的递归函数来处理这些啰嗦的工作。

132

#### 4.3.6 平均情况分析

直观地，我们期望前一节所有的操作特别是makeEmpty和operator=都花费 $O(\log N)$ 时间，因为我们用常数时间在树中降低了一层，这样一来，对树的操作大致减少一半左右。实际上，所有操作（除了makeEmpty和operator=）的运行时间都是 $O(d)$ ，其中d是包含所访问项的结点的深度。



```

1  /**
2   * Destructor for the tree
3   */
4  ~BinarySearchTree( )
5  {
6      makeEmpty( );
7  }
8  /**
9   * Internal method to make subtree empty.
10 */
11 void makeEmpty( BinaryNode * & t )
12 {
13     if( t != NULL )
14     {
15         makeEmpty( t->left );
16         makeEmpty( t->right );
17         delete t;
18     }
19     t = NULL;
20 }

```

图4-27 析构函数和递归makeEmpty成员函数

```

1  /**
2   * Deep copy.
3   */
4  const BinarySearchTree & operator=( const BinarySearchTree & rhs )
5  {
6      if( this != &rhs )
7      {
8          makeEmpty( );
9          root = clone( rhs.root );
10     }
11     return *this;
12 }
13
14 /**
15  * Internal method to clone subtree.
16  */
17 BinaryNode * clone( BinaryNode *t ) const
18 {
19     if( t == NULL )
20         return NULL;
21
22     return new BinaryNode( t->element, clone( t->left ), clone( t->right ) );
23 }

```

图4-28 operator=和递归的clone成员函数

我们在本节要证明，如果所有的插入序列都是等可能的，那么，树的所有结点的平均深度为  $O(\log N)$ 。

一棵树的所有结点的深度和称为内部路径长 (internal path length)。下面将计算二叉查找树的平均内部路径长，其中的“平均”涵盖二叉查找树中所有可能的插入序列。

令  $D(N)$  是具有  $N$  个结点的某棵树  $T$  的内部路径长， $D(1) = 0$ 。一棵  $N$  结点树是由一棵  $i$  结点左子树和一棵  $(N-i-1)$  结点右子树以及深度为 0 的一个根结点组成，其中  $0 \leq i < N$ ， $D(i)$  为根的左子树的内部路径长。但是在原树中，所有这些结点都要加深一层。同样的结论对于右子树也是成立的。因此我们得到递推关系：

$$D(N) = D(i) + D(N - i - 1) + N - 1$$

133

如果所有子树的大小都是等可能地出现，这对于二叉查找树是成立的（因为子树的大小只依赖于第一个插入到树中的元素的相对的秩（rank）），但对二叉树不成立，那么 $D(i)$ 和 $D(N-i-1)$ 的平均值都是 $(1/N) \sum_{j=0}^{N-1} D(j)$ 。于是

$$D(N) = \frac{2}{N} \left[ \sum_{j=0}^{N-1} D(j) \right] + N - 1$$

在第7章将遇到并求解这个递推关系，得到的平均值为 $D(N) = O(N \log N)$ 。因此任意结点预期的深度为 $O(\log N)$ 。作为一个例子，图4-29所示随机生成的500个结点的树的结点期望深度为9.98。

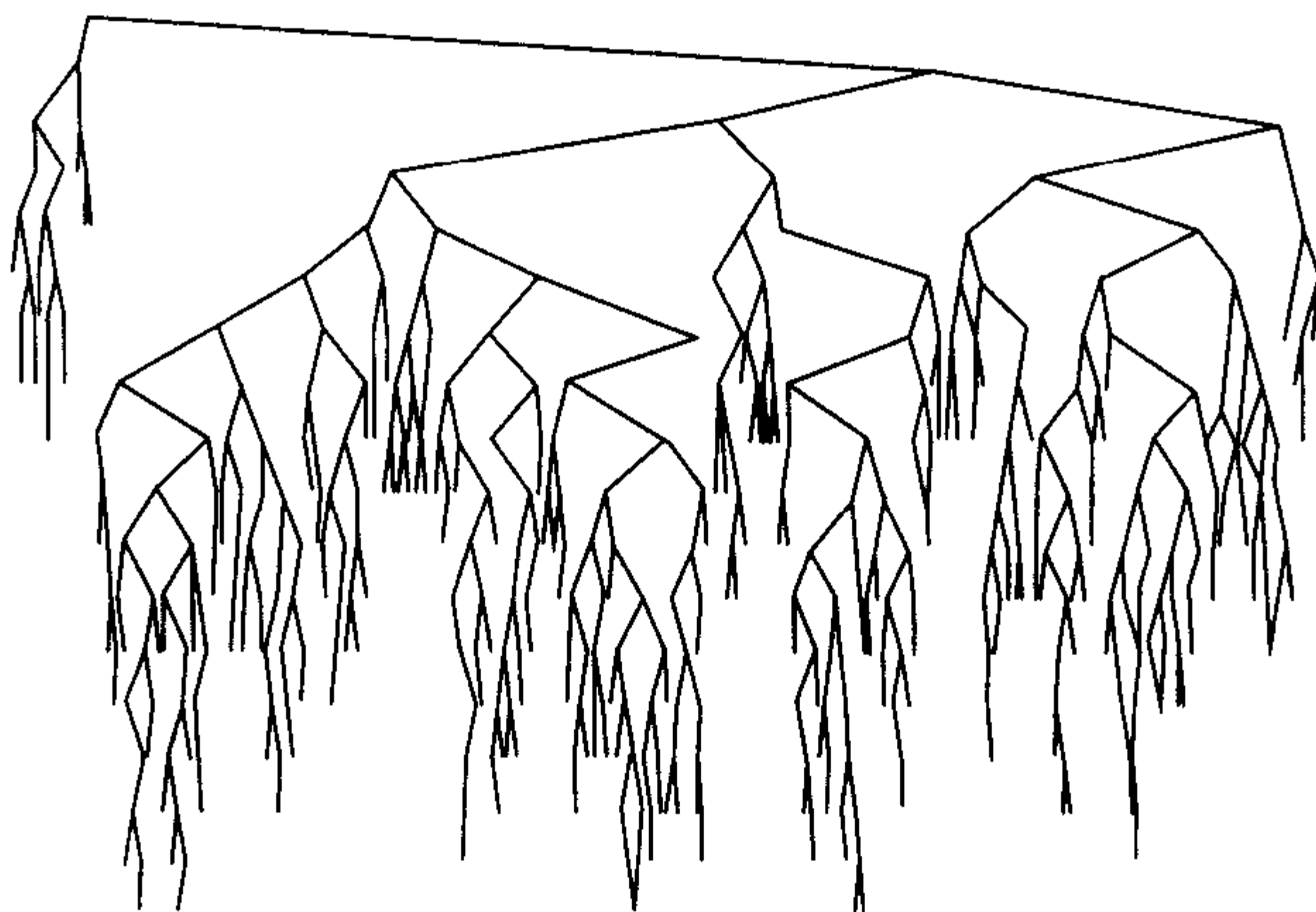
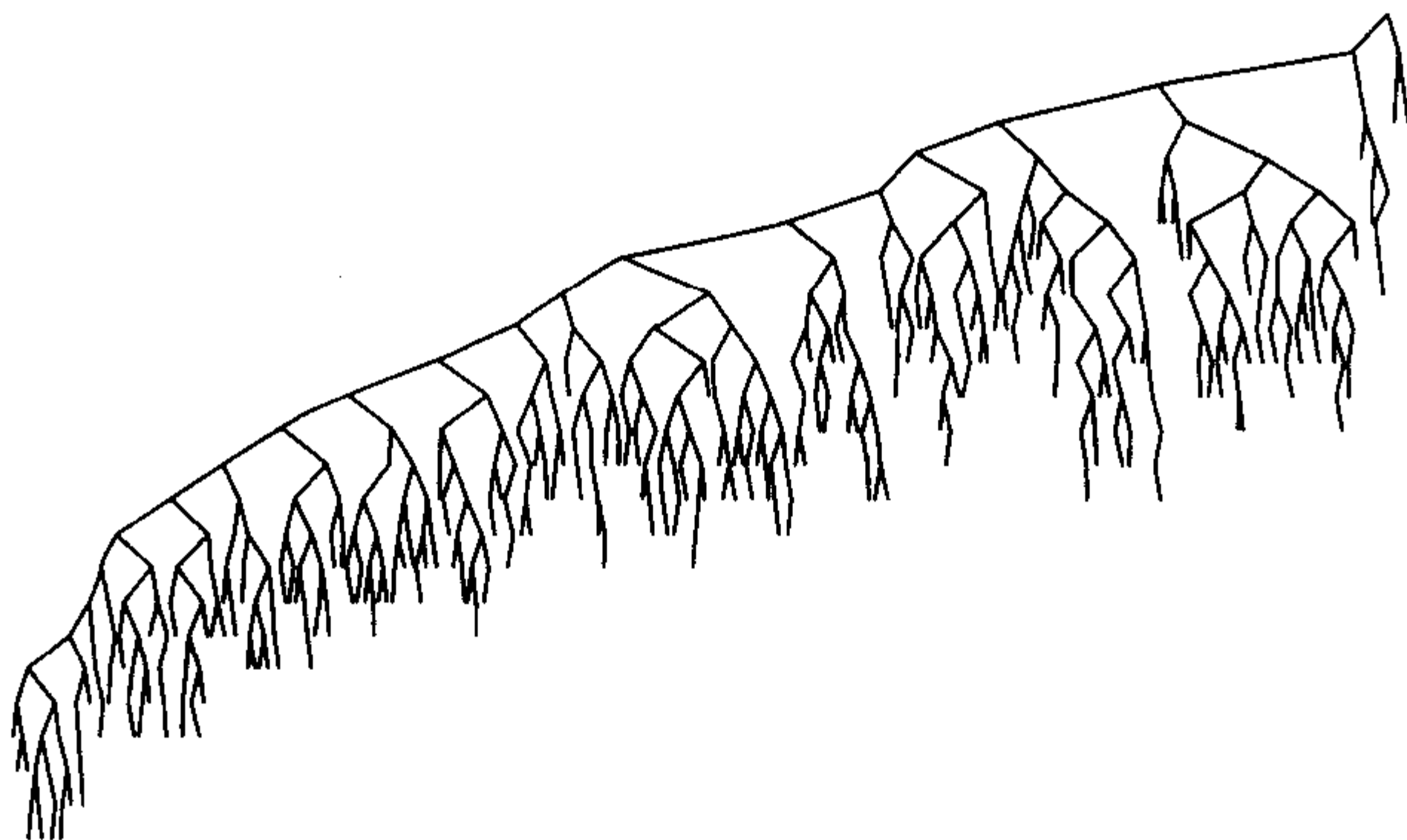


图4-29 一棵随机生成的二叉查找树

但是，一上来就断言这个结果意味着上一节讨论的所有操作的平均运行时间是 $O(\log N)$ 并不完全正确。原因在于删除操作，我们并不清楚是否所有的二叉查找树都是等可能出现的。特别是，上面描述的删除算法有助于使得左子树比右子树深度深，因为我们总是用右子树的一个结点来代替删除的结点。这种方法的准确效果仍然是未知的，但它似乎只是理论上的谜团。已经证明，如果交替插入和删除 $\Theta(N^2)$ 次，那么树的期望深度将是 $\Theta(\sqrt{N})$ 。在25万次随机insert/remove操作后，图4-29中右沉的树看起来明显地不平衡（平均深度=12.51），见图4-30。

134

图4-30 在 $\Theta(N^2)$ 次insert/remove操作后的二叉查找树

135

在删除操作中，可以通过随机选取右子树的最小元素或左子树的最大元素来代替被删除的元素以消除这种不平衡问题。这种做法显然消除了上述偏向并使树保持平衡，但是，没有人实际上证明过这一点。无论如何，这种现象似乎主要是理论上的问题，因为对于小的树，上述效果根本显示不出来，甚至更奇怪，如果使用 $o(N^2)$ 次insert/remove对操作，那么树似乎可以得到平衡！

上面的讨论主要是说明，明确“平均”意味着什么，一般是极其困难的，可能需要一些假设，这些假设可能合理，也可能不合理。不过，在没有删除或是使用懒惰删除的情况下，可以证明所有二叉查找树都是等可能出现的，而且可以断言：上述操作的平均运行时间都是 $O(\log N)$ 。除像上面讨论的一些个别情形外，这个结果与实际观察到的情形是非常吻合的。

如果向一棵预先排序的树输入数据，那么一连串insert操作将花费二次的时间，而链表实现的代价会非常巨大，因为此时的树将只由那些没有左儿子的结点组成。一种解决办法就是要有一个称为平衡（balance）的附加的结构条件：任何结点的深度均不得过深。

有许多一般的算法实现平衡树。但是，大部分算法都要比标准的二叉查找树复杂得多，而且更新要平均花费更长的时间。不过，它们确实防止了处理起来非常麻烦的一些简单情形。下面，将介绍最老的一种平衡查找树，即AVL树。

另外，较新的方法是放弃平衡条件，允许树有任意的深度，但是在每次操作之后要使用一个调整规则进行调整，使得后面的操作效率更高。这种类型的数据结构一般属于自调整（self-adjusting）类结构。在二叉查找树的情况下，对于任意单个运算我们不再保证 $O(\log N)$ 的时间界，但是可以证明任意连续 $M$ 次操作在最坏的情形下花费时间 $O(M \log N)$ 。一般这足以防止令人棘手的最坏情形。我们将要讨论的数据结构叫作伸展树（splay tree）；它的分析相当复杂，将在第11章讨论。

## 4.4 AVL树

AVL（Adelson-Velskii and Landis）树是带有平衡条件（balance condition）的二叉查找树。这个平衡条件必须要容易保持，而且必须保证树的深度是 $O(\log N)$ 。最简单的想法是要求左右子树具有相同的高度。如图4-31所示，这种想法并不强求树的深度要浅。

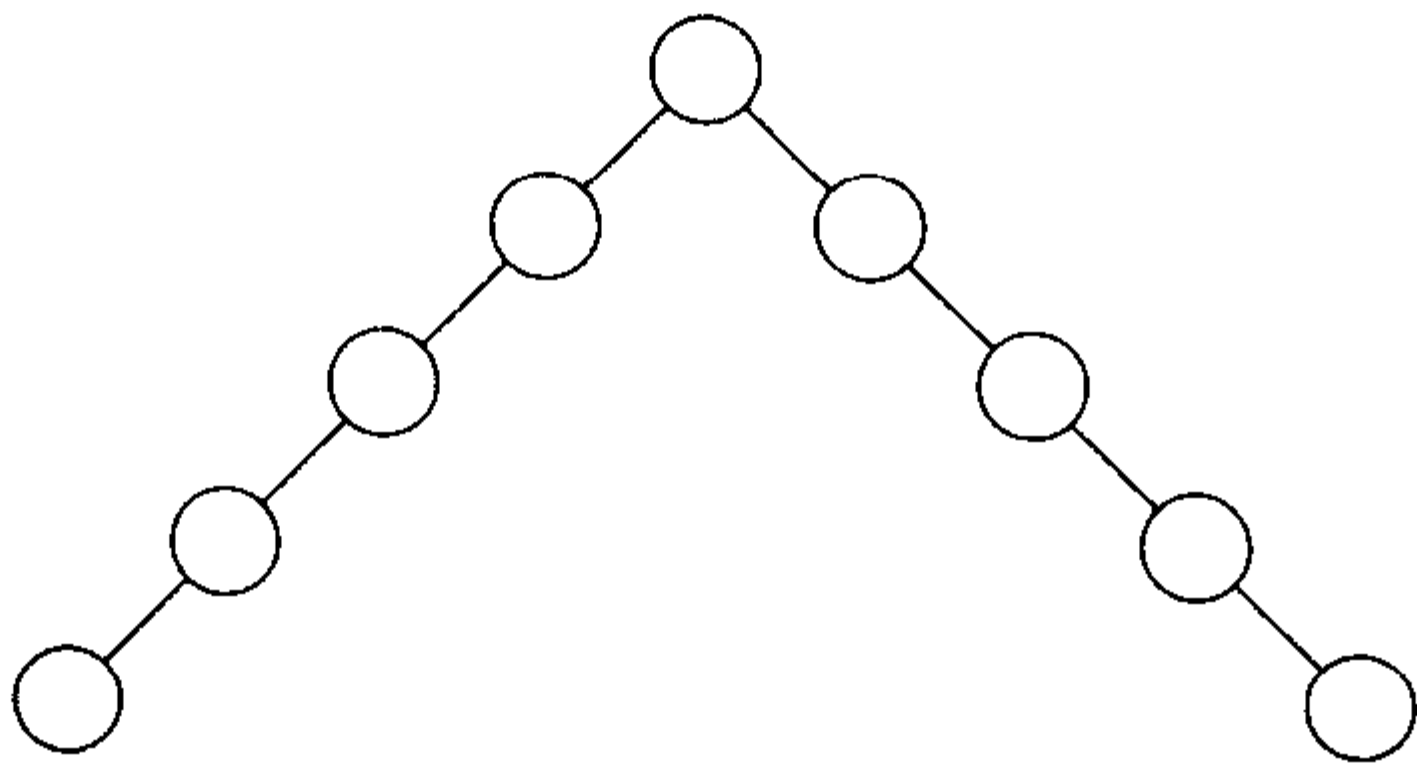


图4-31 一棵坏的二叉树。仅要求在根结点平衡是不够的

另一种平衡条件是要要求每个结点都必须有相同高度的左子树和右子树。如果空子树的高度定义为-1（通常就是这么定义），那么只有具有 $2^k - 1$ 个结点的理想平衡树满足这个条件。因此，虽然这种平衡条件保证了树的深度小，但是它太严格，难以使用，需要放宽条件。

一棵AVL树是其每个结点的左子树和右子树的高度最多差1的二叉查找树（空树的高度定义为-1）。在图4-32中，左边的树是AVL树，但是右边的树不是。每一个结点（在其结点结构中）保留高度信息。可以证明，大致上讲，一棵AVL树的高度最多为 $1.44 \log(N + 2) - 1.328$ ，但是实

际上的高度只比 $\log N$ 稍微多一些。作为例子，图4-33显示了一棵具有最少结点（143）高度为9的AVL树。这棵树的左子树是高度为7且结点数最少的AVL树，右子树是高度为8且结点数最少的AVL树。它告诉我们，在高度为 $h$ 的AVL树中，最少结点数 $S(h)$ 由 $S(h) = S(h-1) + S(h-2) + 1$ 给出。对于 $h=0$ ， $S(h)=1$ ； $h=1$ ， $S(h)=2$ 。函数 $S(h)$ 与斐波那契数密切相关，由此推出上面提到的关于AVL树的高度的界。

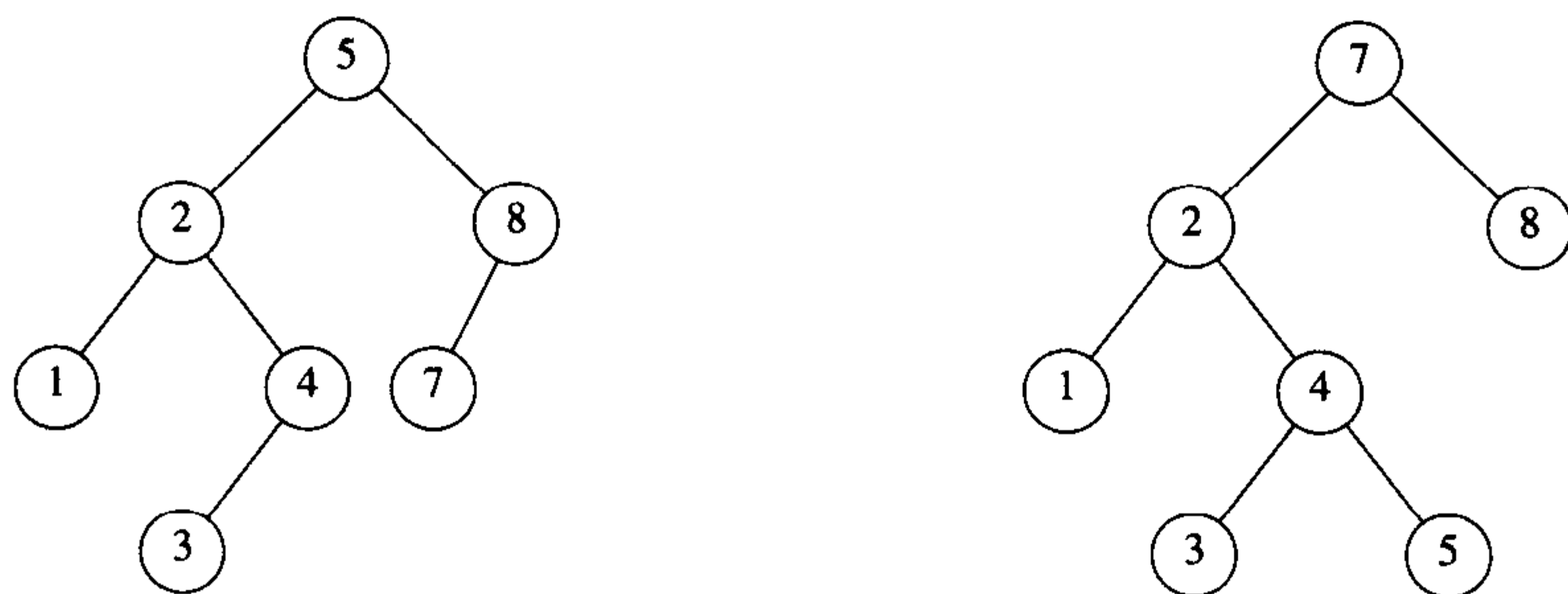


图4-32 两棵二叉查找树，只有左边的树是AVL树

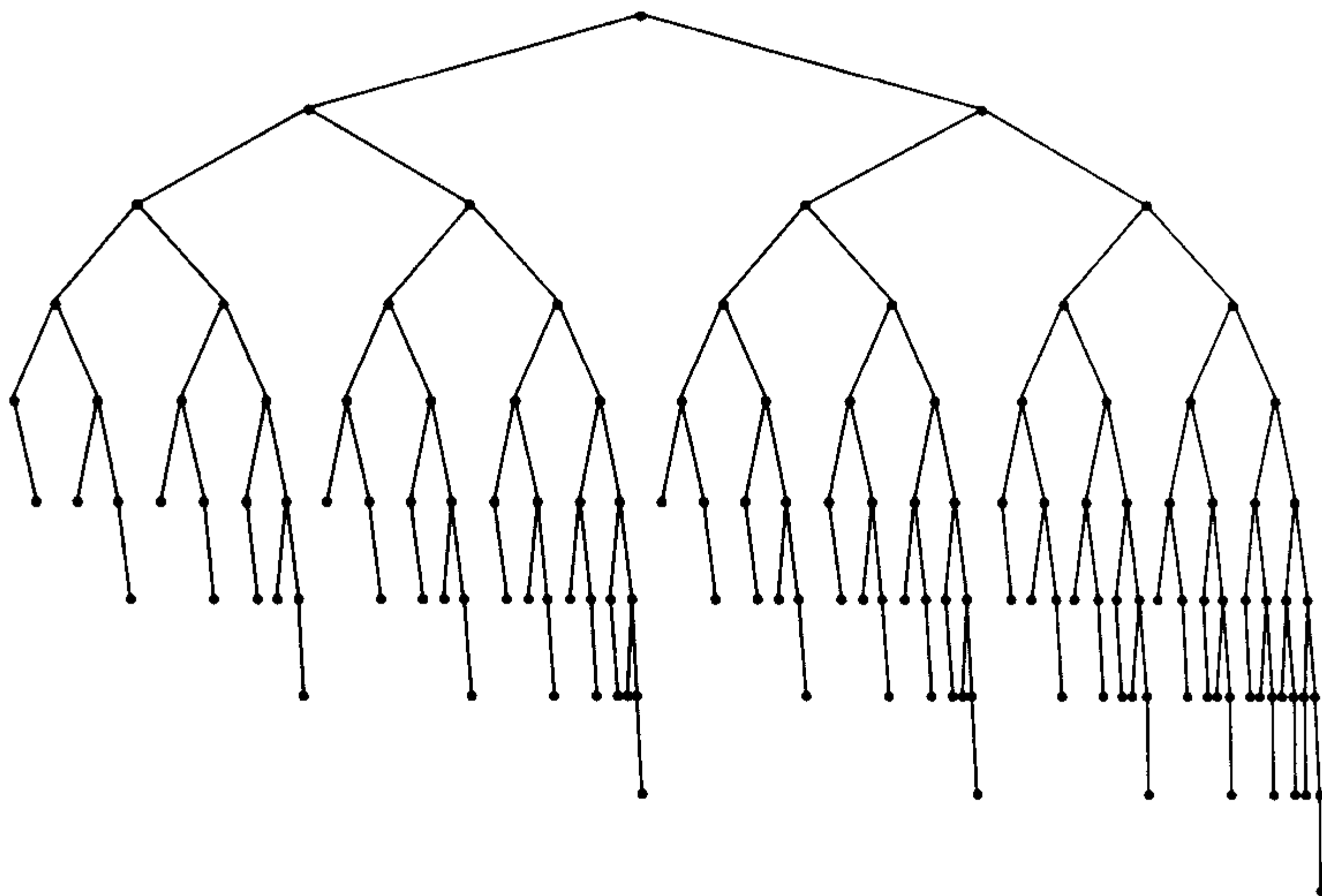


图4-33 高度为9的最小的AVL树

因此，除去可能的插入外（我们将假设懒惰删除），所有的树操作都可以以时间 $O(\log N)$ 执行。当进行插入操作时，需要更新通向根结点路径上那些结点的所有平衡信息，而插入操作隐含着困难的原因在于，插入一个结点可能破坏AVL树的特性（例如，将6插入到图4-32中的AVL树中将会破坏根为8的结点平衡条件）。如果发生这种情况，那么就要恢复平衡的性质后才认为这一步插入操作完成。事实上，这总可以通过对树进行简单的修正来做到，我们称其为**旋转**（rotation）。

在插入以后，只有那些从插入点到根结点的路径上的结点的平衡可能被改变，因为只有这些结点的子树可能发生变化。当我们沿着这条路径上行到根并更新平衡信息时，可以发现一个结点，它的新平衡破坏了AVL条件。我们将指出如何在第一个这样的结点（即最深的结点）重新平衡这棵树，并证明这一重新平衡保证整个树满足AVL性质。

把必须重新平衡的结点叫作 $\alpha$ 。由于任意结点最多有两个儿子，因此高度不平衡时， $\alpha$ 点的两棵子树的高度差2。容易看出，这种不平衡可能出现在下面4种情况中：



- (1) 对 $\alpha$ 的左儿子的左子树进行一次插入。
- (2) 对 $\alpha$ 的左儿子的右子树进行一次插入。
- (3) 对 $\alpha$ 的右儿子的左子树进行一次插入。
- (4) 对 $\alpha$ 的右儿子的右子树进行一次插入。

情形(1)和(4)是关于 $\alpha$ 点的镜像对称，而情形(2)和(3)也是关于 $\alpha$ 点的镜像对称。因此，理论上只有两种情况，当然从编程的角度来看还是四种情形。

第一种情况是插入发生在“外边”的情形（即左—左的情况或右—右的情况），该情况通过对树的一次单旋转（single rotation）而完成调整。第二种情况是插入发生在“内部”的情形（即左—右的情况或右—左的情况），该情况通过稍微复杂些的双旋转（double rotation）来处理。我们将会看到，这些都是对树的基本操作，它们多次用于平衡树的一些算法中。本节其余部分将描述这些旋转，证明它们足以保持树的平衡，并顺便给出AVL树的一种非正式实现方法。第12章描述其他的平衡树方法，它们着眼于AVL树的更仔细的实现。

#### 4.4.1 单旋转

图4-34显示了单旋转如何调整情形(1)。旋转前的图在左边，而旋转后的图在右边。让我们来具体分析的做法。结点 $k_2$ 不满足AVL平衡性质，因为它的左子树比右子树深2层（图中中间的几条虚线表示树的各层）。该图所描述的情况只是情形(1)的一种可能的情况，在插入之前 $k_2$ 满足AVL性质，但在插入之后这种性质被破坏了。子树X已经“长”出一层，这使得它比子树Z深出2层。Y不可能与新X在同一层上，因为那样 $k_2$ 在插入以前就已经失去平衡了；Y也不可能与Z在同一层上，因为那样 $k_1$ 就会是在通向根的路径上破坏AVL平衡条件的第一个结点。

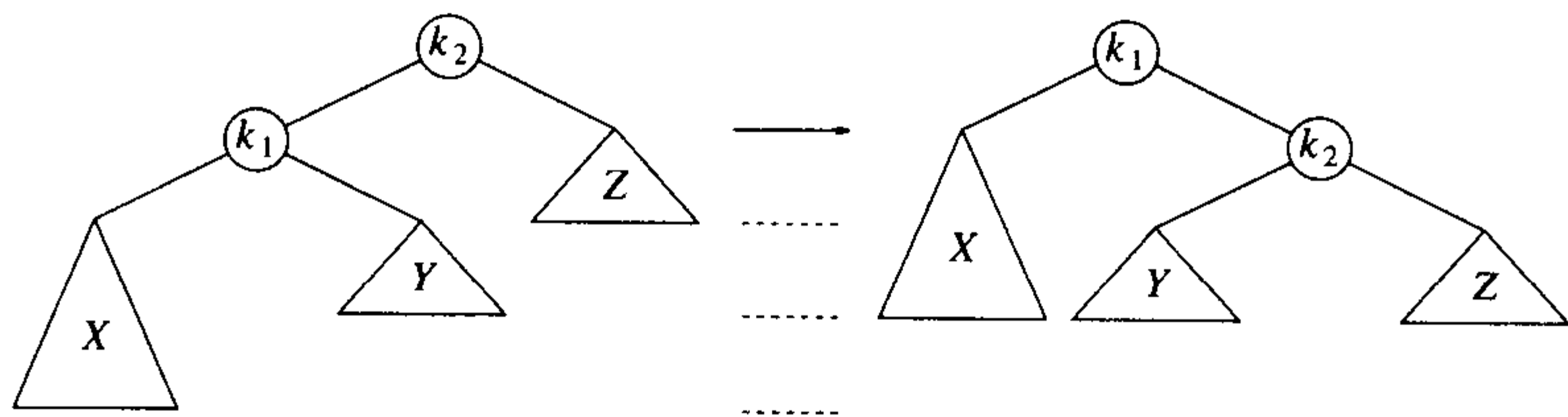


图4-34 单旋转修正情形(1)

为使树恢复平衡，我们把X上移一层，并把Z下移一层。注意，此时实际上超出了AVL性质的要求。为此，重新安排结点以形成一棵等价的树，如图4-34的第二部分所示。抽象地形容就是：把树形象地看成是柔软灵活的，抓住子结点 $k_1$ ，闭上你的双眼，使劲摇动它，在重力作用下， $k_1$ 就变成了新的根。二叉查找树的性质告诉我们，在原树中 $k_2 > k_1$ ，于是在新树中 $k_2$ 变成了 $k_1$ 的右儿子，X和Z仍然分别是 $k_1$ 的左儿子和 $k_2$ 的右儿子。子树Y包含原树中介于 $k_1$ 和 $k_2$ 之间的那些结点，可以将它放在新树中 $k_2$ 的左儿子的位置上，这样，所有对顺序的要求都得到满足。

这样的操作只需要一部分链改变，结果我们得到另外一棵二叉查找树，它是一棵AVL树，因为X向上移动了一层，Y停在原来的水平上，而Z下移一层。 $k_2$ 和 $k_1$ 不仅满足AVL要求，而且它们的子树都恰好处在同一高度上。不仅如此，整个树的新高度恰恰与插入前原树的高度相同，而插入操作却使得子树X长高了。因此，通向根结点的路径的高度不需要进一步的修正，因而也不需要进一步的旋转。图4-35显示了在将6插入左边原始的AVL树后结点8便不再平衡。于是，我们在7和8之间做一次单旋转，结果得到右边的树。

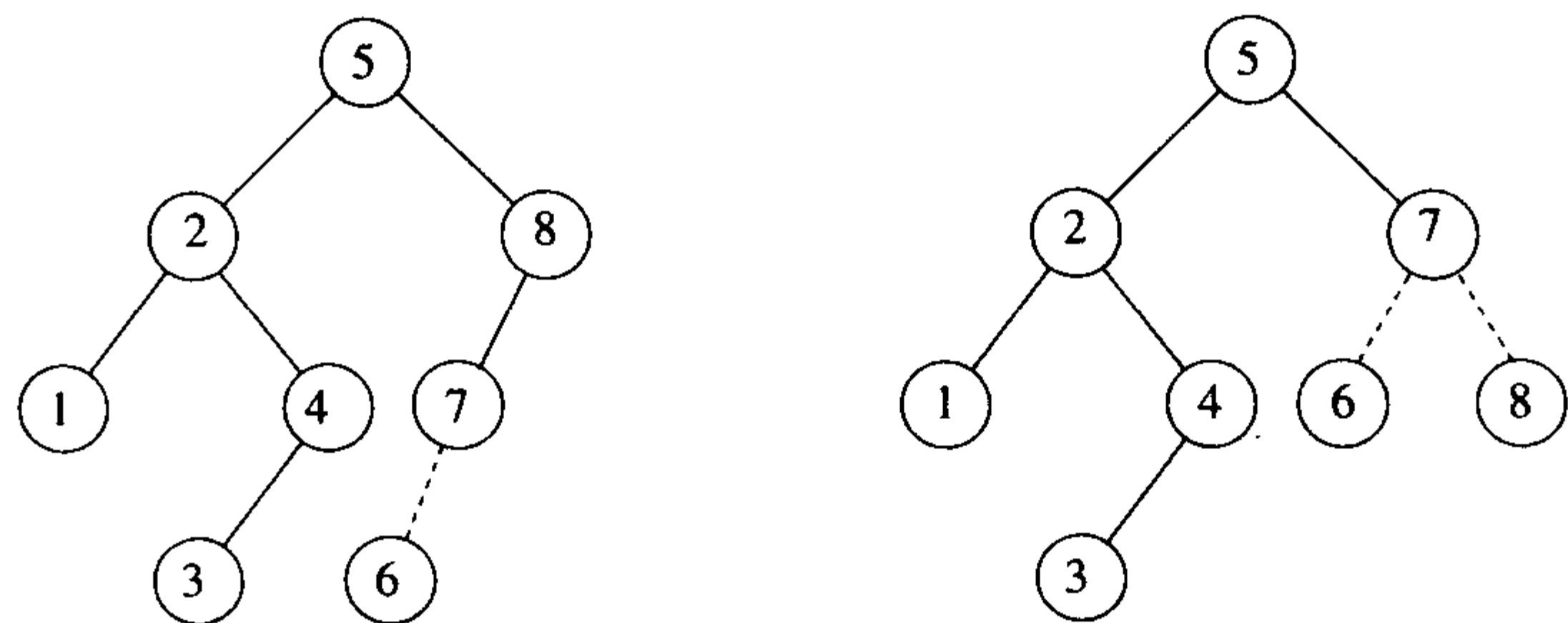


图4-35 插入6破坏了AVL性质，而后经过单旋转又将AVL性质恢复

正如前面所提到的，情形(4)代表一种对称的情形。图4-36指出单旋转如何使用。下面演示一个更长一些的例子。假设从初始的空AVL树开始插入项3、2和1，然后依序插入4到7。在插入项1时第一个问题出现了，AVL性质在根处被破坏。我们在根与其左儿子之间施行单旋转以修正这个问题。下面是旋转之前和之后的两棵树：

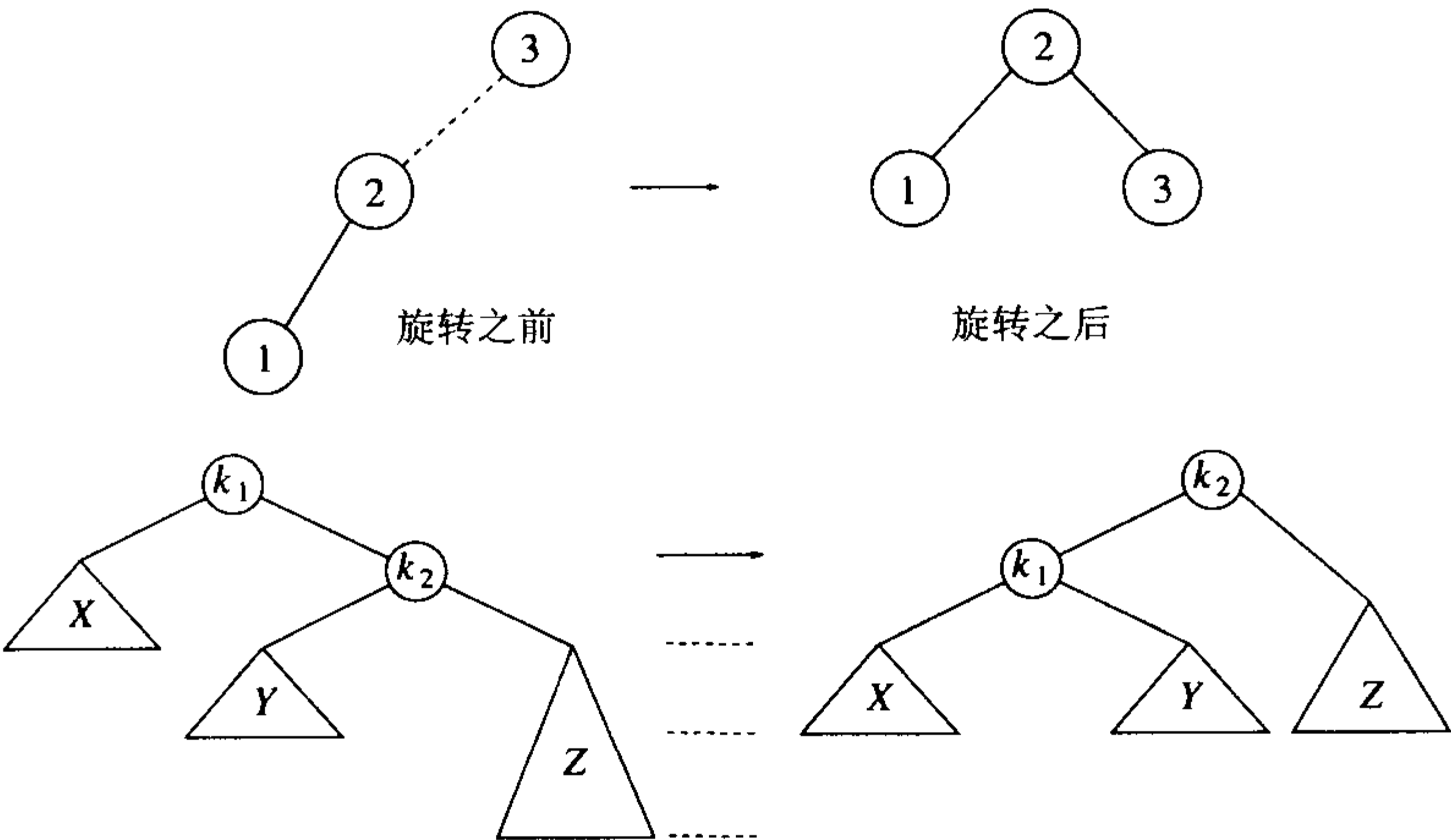
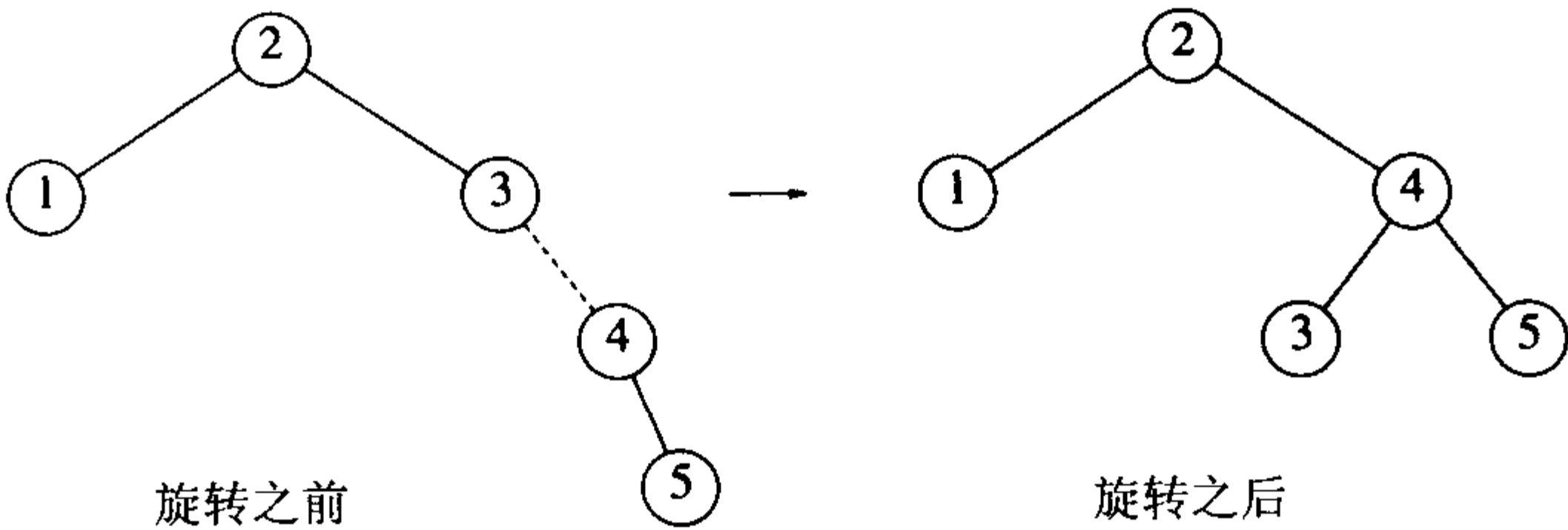
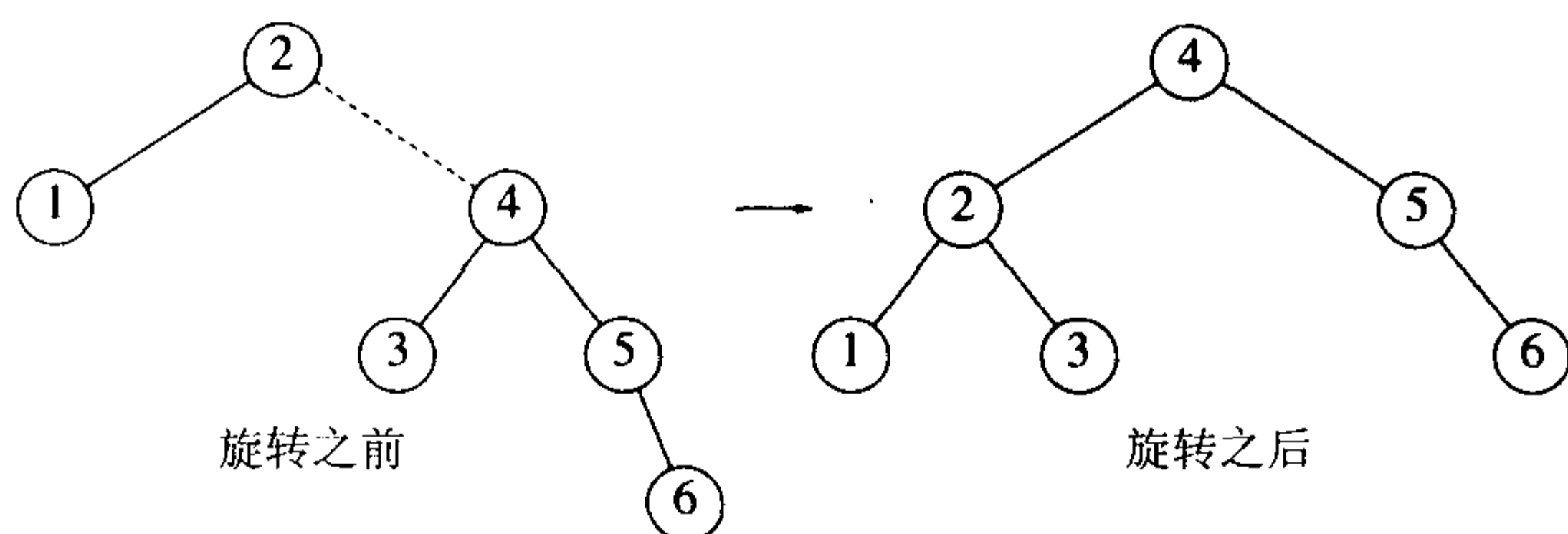


图4-36 单旋转修正情形(4)

图中虚线连接两个结点，它们是旋转的主体。下面我们插入4，这没有问题，但插入5破坏了结点3处的AVL性质，而通过单旋转又将其修正。除旋转引起的局部变化外，编程人员必须记住：树的其余部分必须知晓该变化。如本例中，结点2的右儿子必须重新设置以链接到4而不是3。这一点很容易忘记，从而导致树被破坏（4就会是不可访问的）。 140

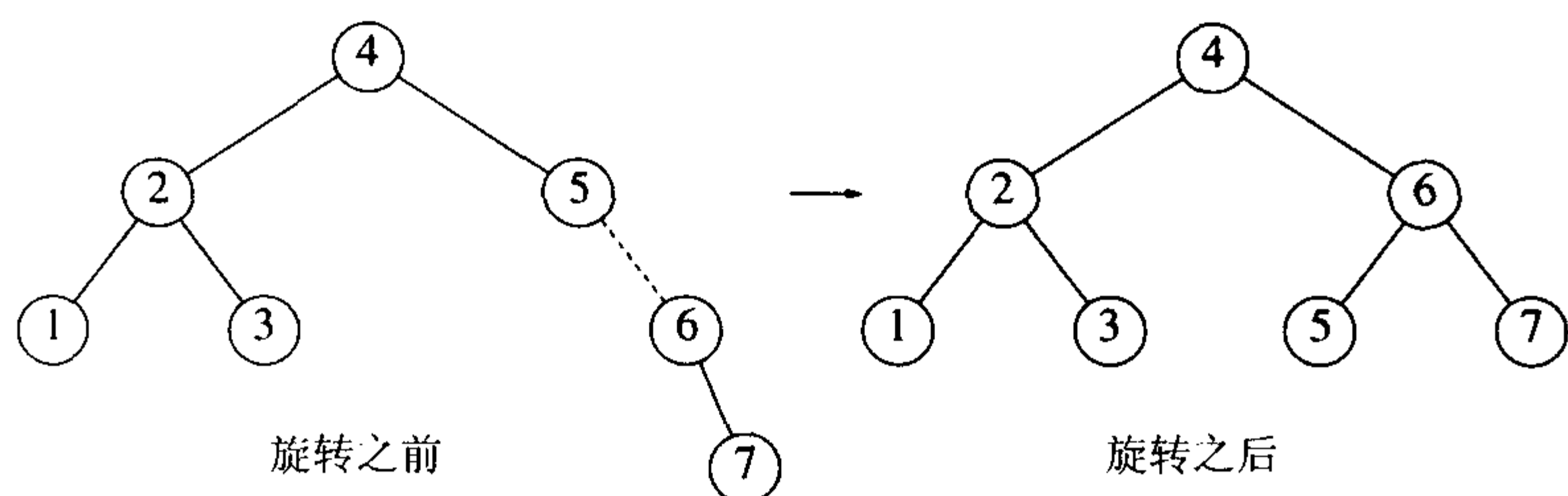


下面我们插入6。这在根结点产生一个平衡问题，因为它的左子树高度是0而右子树高度为2。因此我们在根处的2和4之间实施一次单旋转。



141

旋转的结果使得2是4的一个儿子而4原来的左子树变成结点2的新的右子树。在该子树上的每一项均在2和4之间，因此这个变换是成立的。我们插入的下一项是7，它导致另一次旋转：



#### 4.4.2 双旋转

上面描述的算法有一个问题：如图4-37所示，它对于情形(2)和(3)上面的做法无效。问题在于子树Y太深，单旋转没有减低它的深度。解决这个问题的双旋转在图4-38中表示出。

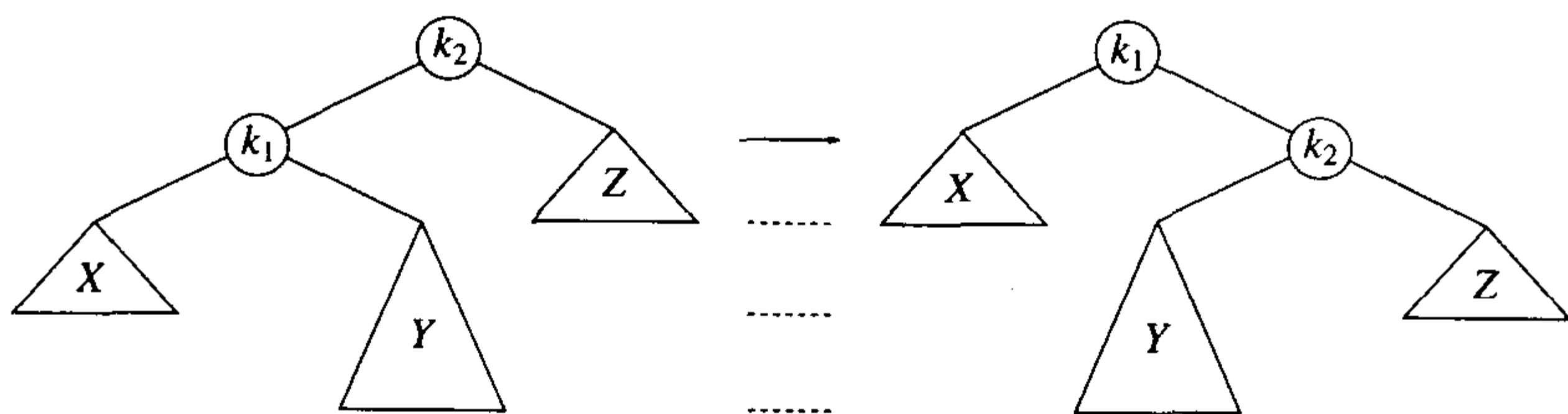


图4-37 单旋转不能修正情形(2)

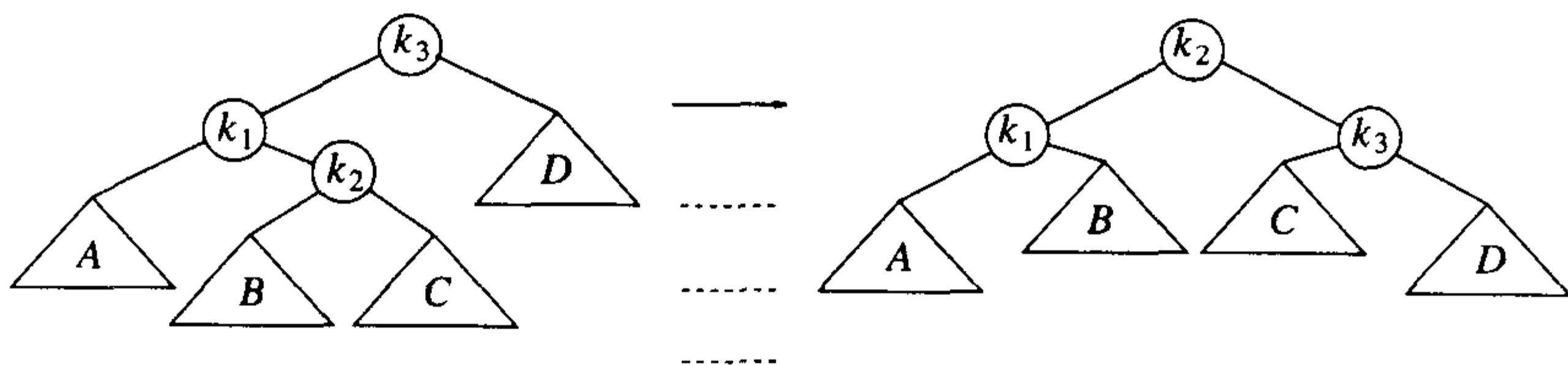


图4-38 左-右双旋转修正情形(2)

在图4-37中，子树Y已经有一项插入其中，这个事实保证它是非空的。因此，可以假设它有一个根和两棵子树。于是，可以把整棵树看成是四棵子树由3个结点连接。如图所示，恰好树B或树C中有一棵比D深两层（除非它们都是空的），但是不能肯定是哪一棵。事实上这无关紧要，在图4-38中B和C都被画成比D低 $1\frac{1}{2}$ 层。

为了重新平衡，我们看到，不能再让 $k_3$ 作为根了，而图4-37所示的在 $k_3$ 和 $k_1$ 之间的旋转又解决不了问题，唯一的选择就是把 $k_2$ 作为新的根。这迫使 $k_1$ 是 $k_2$ 的左儿子， $k_3$ 是它的右儿子，从而完全

确定了这4棵树的最终位置。容易看出,最后得到的树满足AVL树的性质,与单旋转的情形一样,我们也把树的高度恢复到插入以前的水平,这可保证所有的重新平衡和高度更新是完善的。图4-39指出,对称情形(3)也可以通过双旋转得以修正。在这两种情形下,其效果与先在 $\alpha$ 的儿子和孙子之间旋转而后再在 $\alpha$ 和它的新儿子之间旋转的效果是相同的。

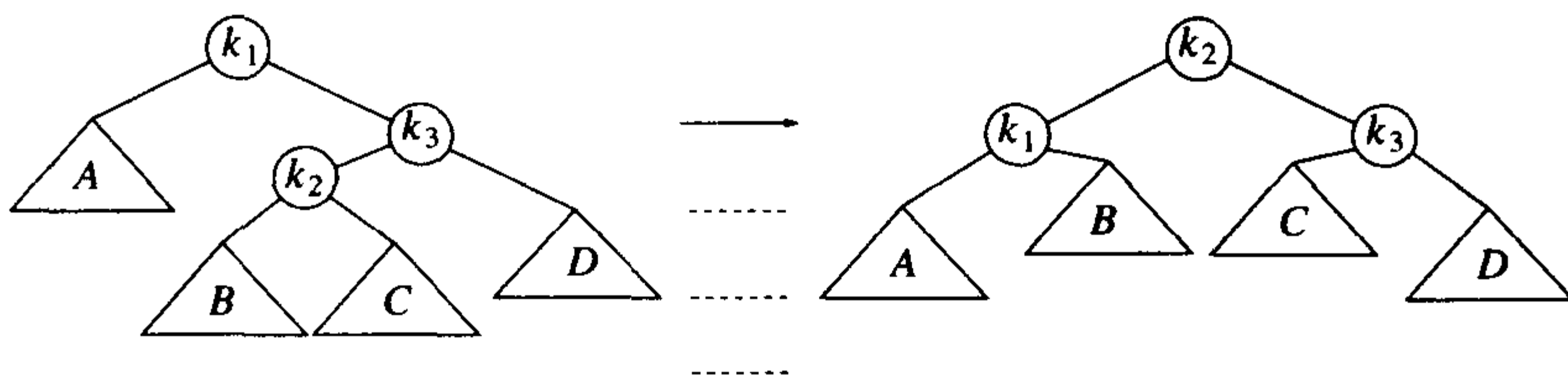
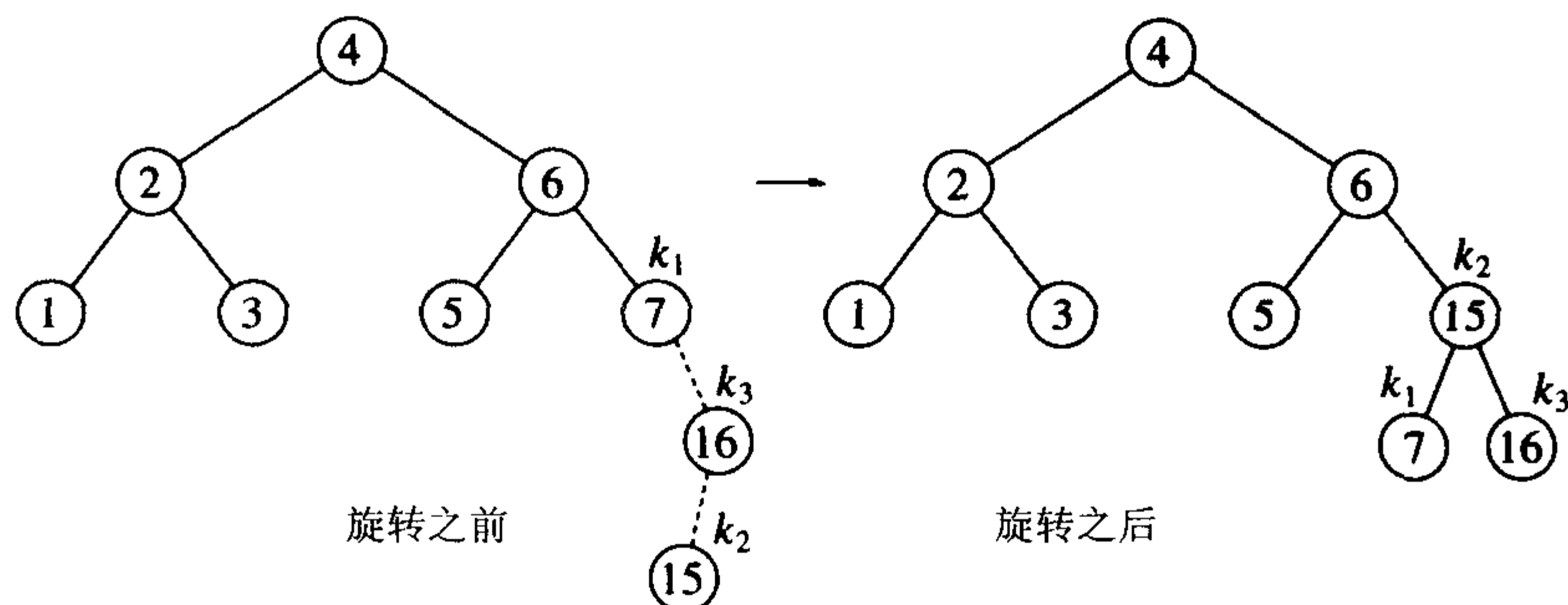


图4-39 右-左双旋转修正情形(3)

我们继续在前面例子的基础上以倒序插入10-16,接着插入8,然后再插入9。插入16很容易,因为它并不破坏平衡性质,但是插入15会引起结点7处的高度不平衡。这属于情形(3),需要通过一次右-左双旋转来解决。在我们的例子中,这个右-左双旋转将涉及7、16和15。此时, $k_1$ 是具有项7的结点, $k_3$ 是具有项16的结点,而 $k_2$ 是具有项15的结点。子树A、B、C和D都是空树。

142

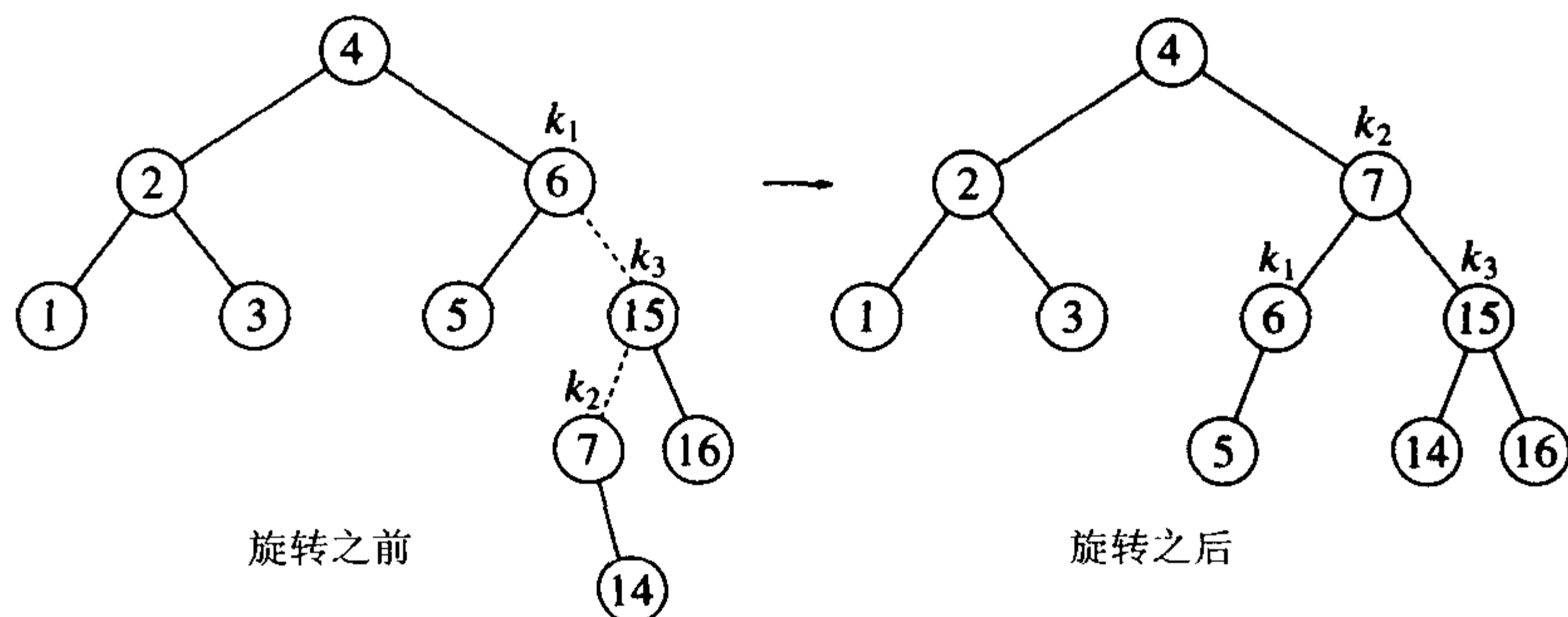


旋转之前

旋转之后

下面我们插入14,它也需要一个双旋转。此时修正该树的双旋转还是右-左双旋转,它将涉及6、15和7。在这种情况下, $k_1$ 是具有项6的结点, $k_2$ 是具有项7的结点,而 $k_3$ 是具有项15的结点。子树A的根在项为5的结点上,子树B是空子树,它是项7的结点原先的左儿子,子树C置根于项14的结点上,最后,子树D的根在项为16的结点上。

143

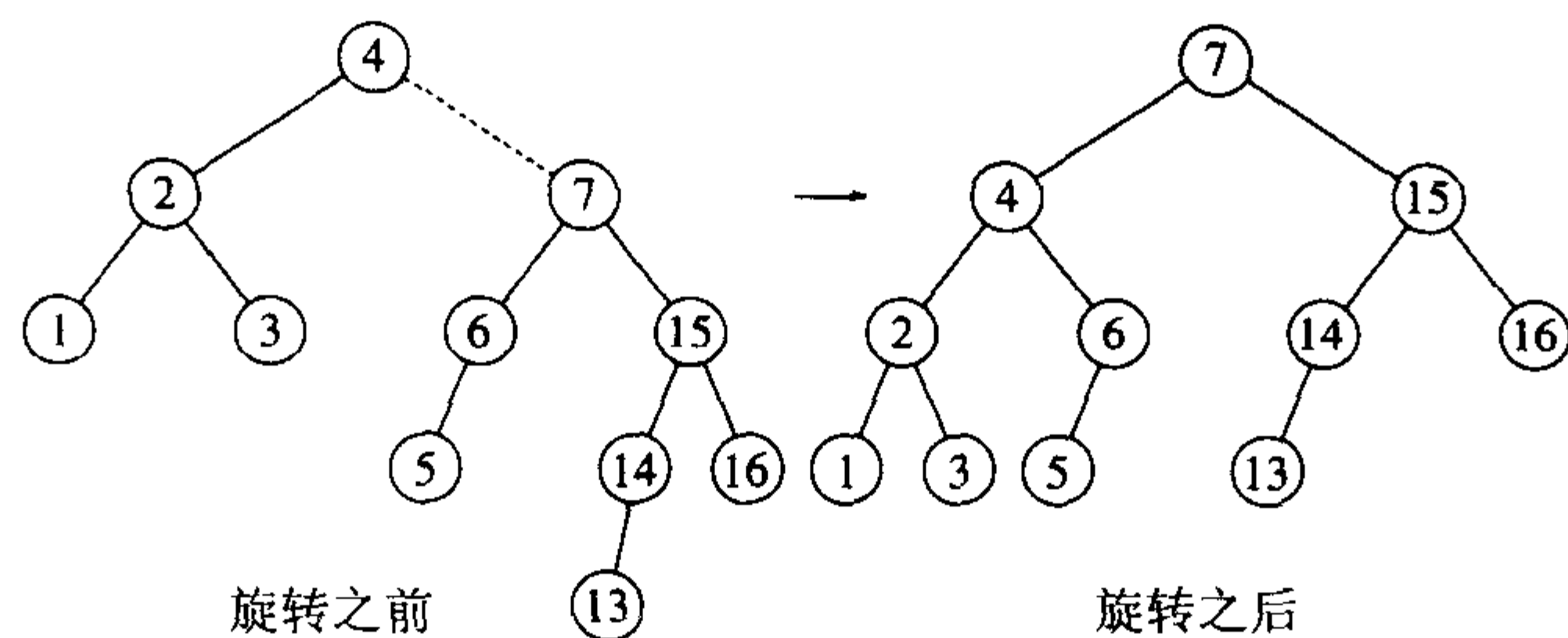


旋转之前

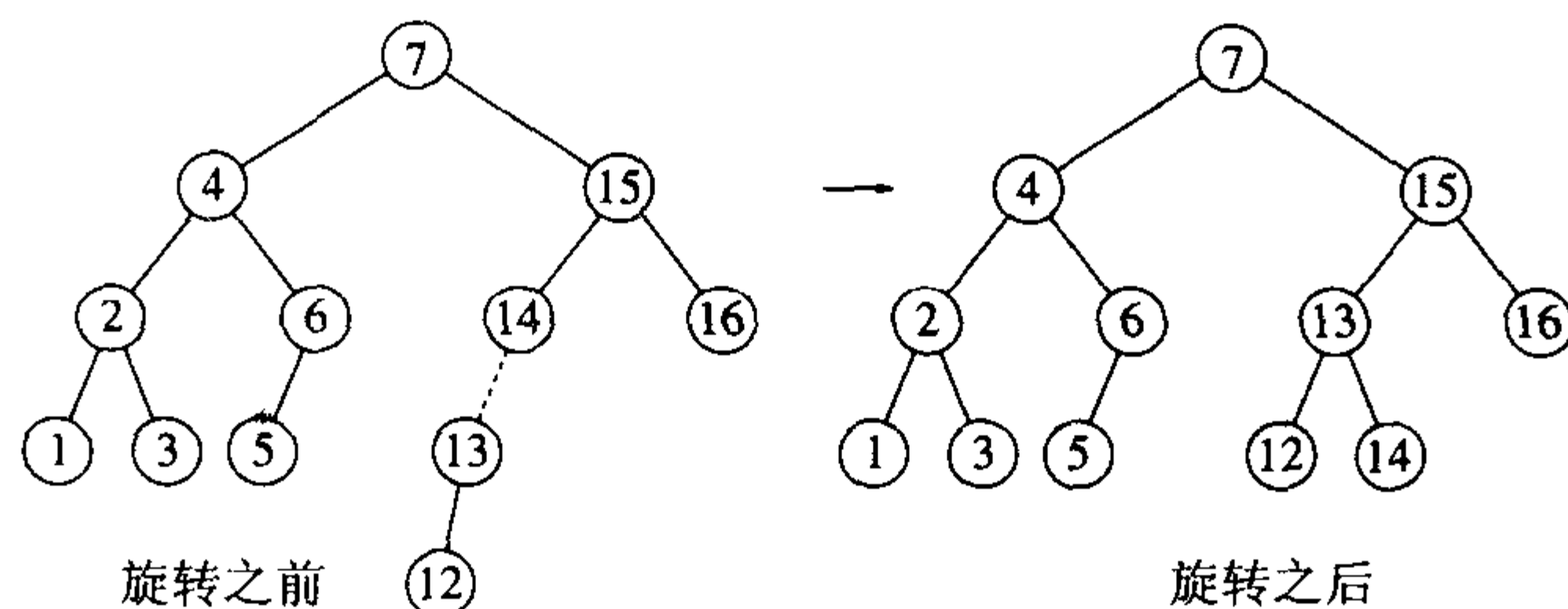
旋转之后

如果现在插入13,那么在根处就会产生不平衡。由于13不在4和7之间,因此我们知道一次单旋转就能完成修正的工作。

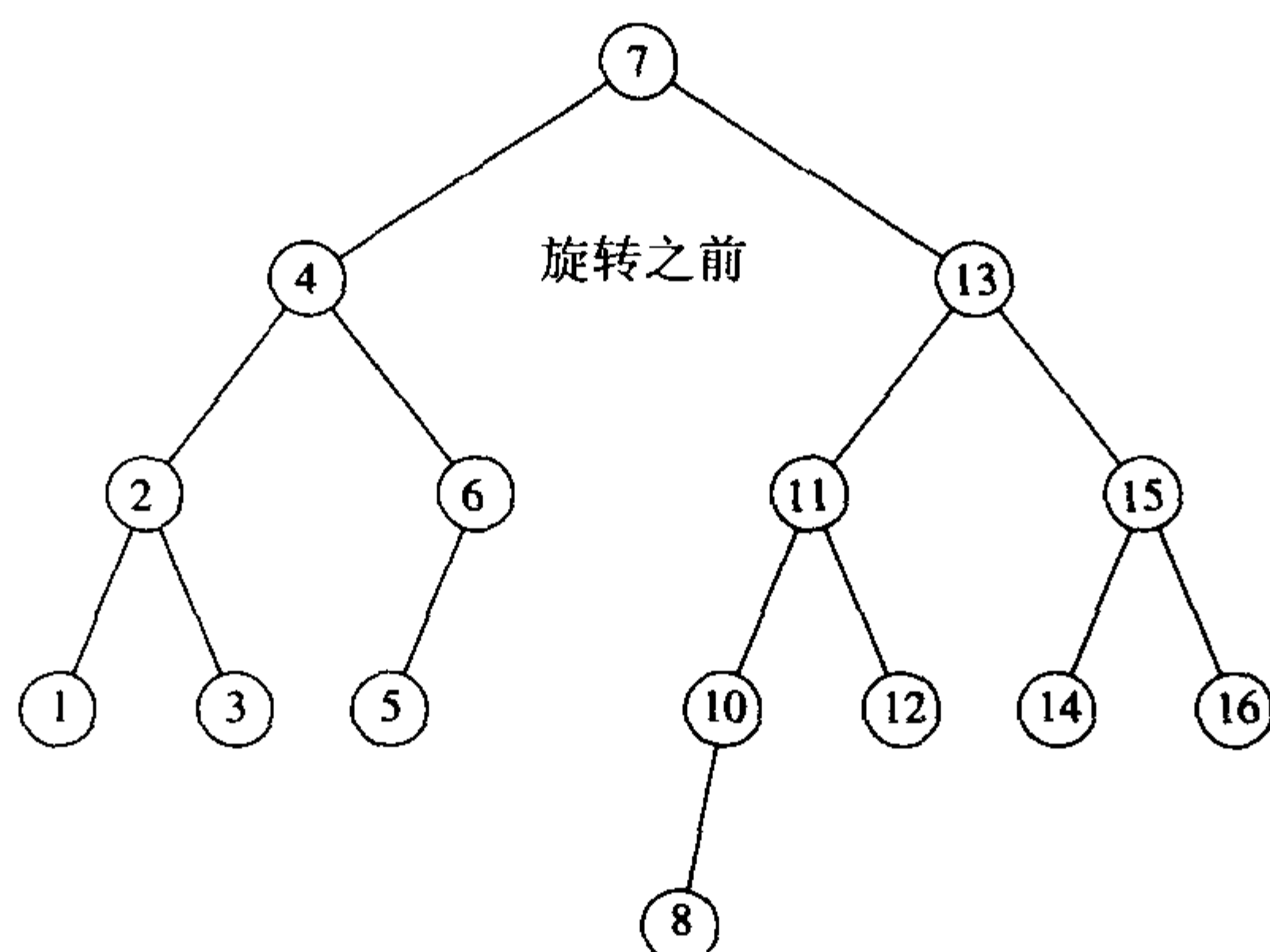




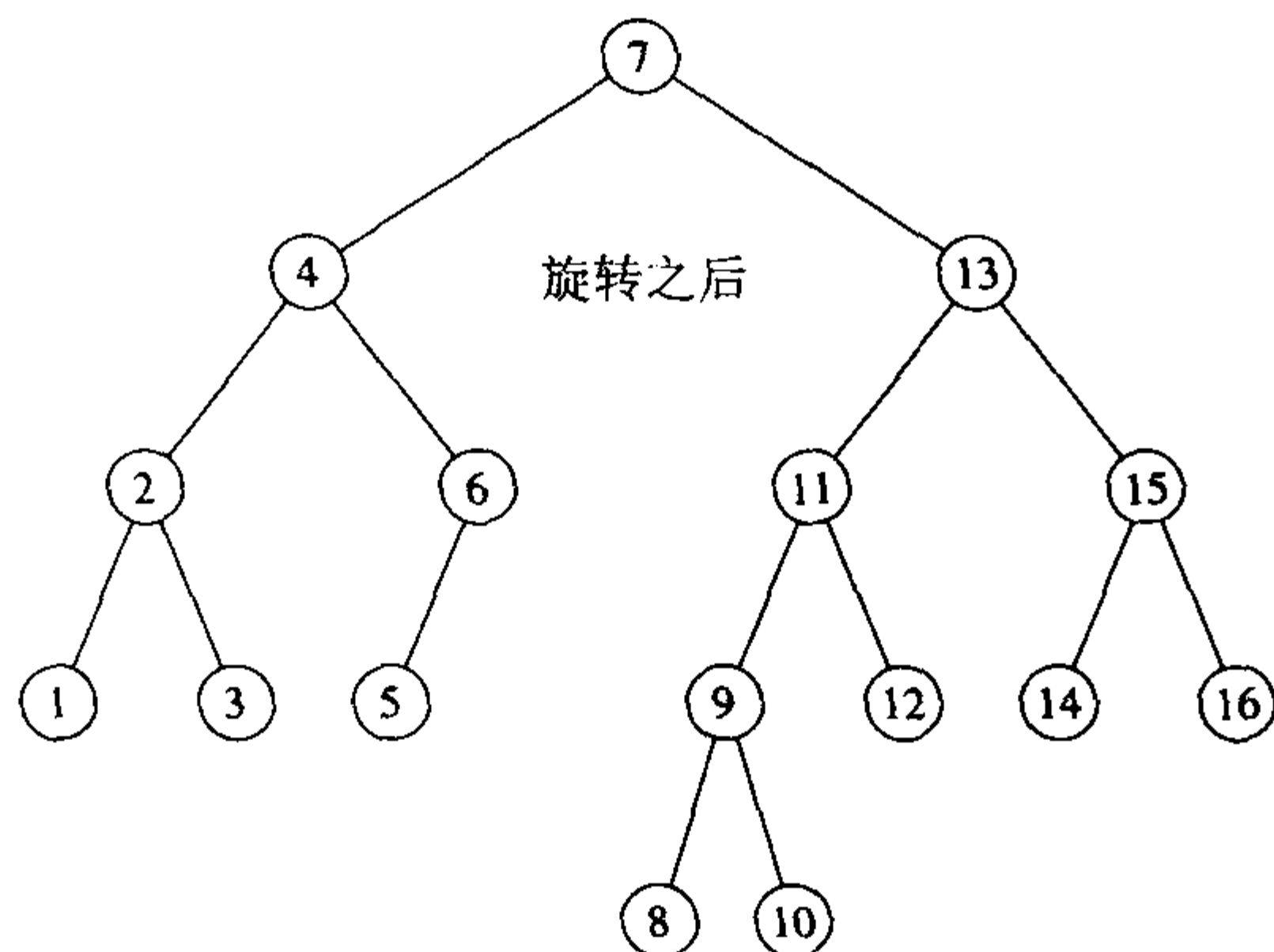
12的插入也需要一个单旋转：



为了插入11，还需要进行一个单旋转，对于其后的10的插入也需要这样的旋转。我们插入8  
 144 而不进行旋转，这样就建立了一棵近乎理想的平衡树。



最后，插入9以演示双旋转的对称情形。注意，9引起含有10的结点产生不平衡。由于9在10和8之间（8是通向9的路径上的结点10的儿子），因此需要进行一个双旋转，我们得到下面的树：



现在对上面的讨论做个总结。除几种情形外，编程的细节是相当简单的。要将项为 $X$ 的一个新结点插入到一棵AVL树 $T$ 中去，我们递归地将 $X$ 插入到 $T$ 的相应的子树（称为 $T_{LR}$ ）中。如果 $T_{LR}$ 的高度不变，那么插入完成。否则，如果在 $T$ 中出现高度不平衡，那么根据 $X$ 以及 $T$ 和 $T_{LR}$ 中的项做适当的单旋转或双旋转，更新这些高度（并解决好与树的其余部分的链接），从而完成插入。由于一次旋转总能足以解决问题，因此仔细地编写非递归的程序一般说来要比编写递归程序快很多。然而，要想把非递归程序编写正确是相当困难的，因此许多编程人员还是用递归的方法实现AVL树。

145

另一个效率问题涉及高度信息的存储。由于真正需要的实际上就是子树高度的差，我们希望确保它很小。如果我们真的尝试这种方法，则可用两个二进制位（代表+1、0、-1）表示这个差。这么做将避免平衡因子的重复计算，但是却丧失了某些简明性。最后的程序或多或少要比在每一个结点存储高度复杂。如果编写递归程序，那么速度恐怕不是主要考虑的问题。此时，通过存储平衡因子所得到的些许速度优势很难抵消清晰和相对简明性的损失。不仅如此，由于大部分机器存储的最小单位是8个二进制位，因此所用的空间量不可能有任何差别。一个8位的char可存储高达127的绝对高度。既然树是平衡的，当然也就不可想像这会不够用（见练习）。

146

有了上面的讨论，现在准备编写AVL树的一些例程。不过，这里只做一部分工作，其余的在线提供。首先，我们需要AvlNode类，它在图4-40中给出。我们还需要一个快速的方法来返回结点的高度，这个方法必须处理NULL指针的恼人的情形。该程序在图4-41中给出。基本的插入例程写起来很容易，因为它主要由一些函数调用组成（见图4-42）。

```

1 struct AvlNode
2 {
3     Comparable element;
4     AvlNode *left;
5     AvlNode *right;
6     int height;
7
8     AvlNode( const Comparable & theElement, AvlNode *lt,
9             AvlNode *rt, int h = 0 )
10    : element( theElement ), left( lt ), right( rt ), height( h )
11 };

```

图4-40 AVL树的结点声明

```

1 /**
2  * Return the height of node t or -1 if NULL.
3  */
4 int height( AvlNode *t ) const
5 {
6     return t == NULL ? -1 : t->height;
7 }

```

图4-41 计算AVL结点高度的函数

```

1 /**
2  * Internal method to insert into a subtree.
3  * x is the item to insert.
4  * t is the node that roots the subtree.
5  * Set the new root of the subtree.
6  */
7 void insert( const Comparable & x, AvlNode * & t )

```

图4-42 向AVL树的插入操作

```

8 {
9     if( t == NULL )
10         t = new AvlNode( x, NULL, NULL );
11     else if( x < t->element )
12     {
13         insert( x, t->left );
14         if( height( t->left ) - height( t->right ) == 2 )
15             if( x < t->left->element )
16                 rotateWithLeftChild( t );
17             else
18                 doubleWithLeftChild( t );
19     }
20     else if( t->element < x )
21     {
22         insert( x, t->right );
23         if( height( t->right ) - height( t->left ) == 2 )
24             if( t->right->element < x )
25                 rotateWithRightChild( t );
26             else
27                 doubleWithRightChild( t );
28     }
29     else
30         ; // Duplicate; do nothing
31     t->height = max( height( t->left ), height( t->right ) ) + 1;
32 }

```

图4-42 向AVL树的插入操作 (续)

对于图4-43中的树，函数rotateWithRightChild把左边的树变成右边的树，并返回指向新根的指针。函数rotateWithRightChild是对称的。程序在图4-44中示出。

147

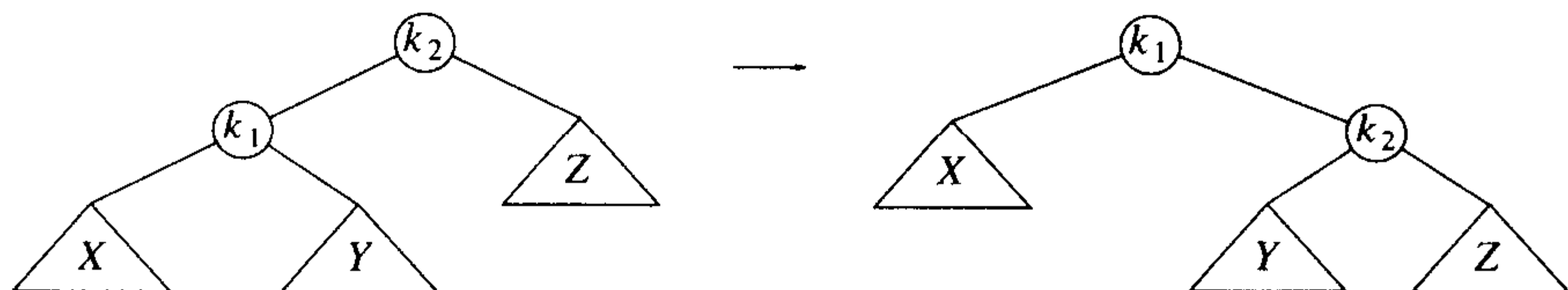


图4-43 单旋转

```

1 /**
2  * Rotate binary tree node with left child.
3  * For AVL trees, this is a single rotation for case 1.
4  * Update heights, then set new root.
5  */
6 void rotateWithLeftChild( AvlNode * & k2 )
7 {
8     AvlNode *k1 = k2->left;
9     k2->left = k1->right;
10    k1->right = k2;
11    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
12    k1->height = max( height( k1->left ), k2->height ) + 1;
13    k2 = k1;
14 }

```

图4-44 执行单旋转的例程

我们要写的最后一个方法完成图4-45所描述的双旋转，其程序由图4-46给出。

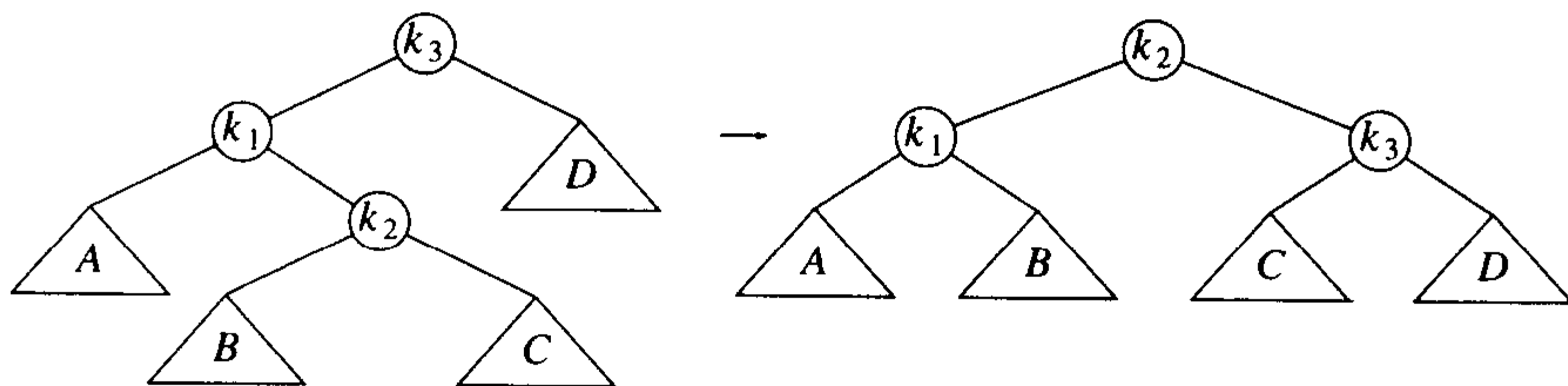


图4-45 双旋转

```

1  /**
2   * Double rotate binary tree node: first left child
3   * with its right child; then node k3 with new left child.
4   * For AVL trees, this is a double rotation for case 2.
5   * Update heights, then set new root.
6   */
7  void doubleWithLeftChild( AvlNode * & k3 )
8  {
9      rotateWithRightChild( k3->left );
10     rotateWithLeftChild( k3 );
11 }

```

图4-46 执行双旋转的例程

对AVL树的删除要比插入稍微复杂一些，我们把它留作练习。如果删除操作相对较少，那么懒惰删除也许是最好的方法。

148

## 4.5 伸展树

本节描述一种相对简单的数据结构，叫作伸展树 (splay tree)，它保证从空树开始任意连续  $M$  次对树的操作最多花费  $O(M \log N)$  时间。虽然这种保证并不排除任意单次操作花费  $O(N)$  时间的可能，而且这样的界也不如每次操作最坏情形的界为  $O(\log N)$  时那么强，但是实际效果却是一样的：不存在不好的输入序列。一般说来，当  $M$  次操作的序列总的最坏情形运行时间为  $O(Mf(N))$  时，我们就说它的摊还 (amortized) 运行时间为  $O(f(N))$ 。因此，一棵伸展树每次操作的摊还代价是  $O(\log N)$ 。经过一系列的操作，有的操作可能花费时间多一些，有的可能要少一些。

伸展树是基于这样的事实：对于二叉查找树来说，每次操作最坏情形时间  $O(N)$  并非不好，只要它相对不常发生就行。任何一次访问，即使花费  $O(N)$ ，仍然可能非常快。二叉查找树的问题在于，虽然一系列访问整体都是不好的操作有可能发生，但是很罕见。此时，累积的运行时间很重要。具有最坏情形运行时间  $O(N)$  但保证对任意  $M$  次连续操作最多花费  $O(M \log N)$  运行时间的查找树数据结构确实可以令人满意了，因为不存在不好的操作序列。

如果任意特定操作可以有最坏时间界  $O(N)$ ，而我们仍然要求一个  $O(\log N)$  的摊还时间界，那么很清楚，只要有一个结点被访问，它就必须被移动。否则，一旦我们发现一个深层的结点，就有可能不断地对它进行访问。如果这个结点不改变位置，而每次访问又花费  $O(N)$ ，那么  $M$  次访问将花费  $O(M \cdot N)$  的时间。

伸展树的基本想法是，当一个结点被访问后，它就要经过一系列AVL树的旋转被推到根上。注意，如果一个结点很深，那么在其路径上就存在许多也相对较深的结点，通过重新构造可以使对所有这些结点的进一步访问所花费的时间变少。因此，如果结点过深，那么重新构造应具有平衡这棵树 (到某种程度) 的作用。除在理论上给出好的时间界外，这种方法还可能有实际的效用，

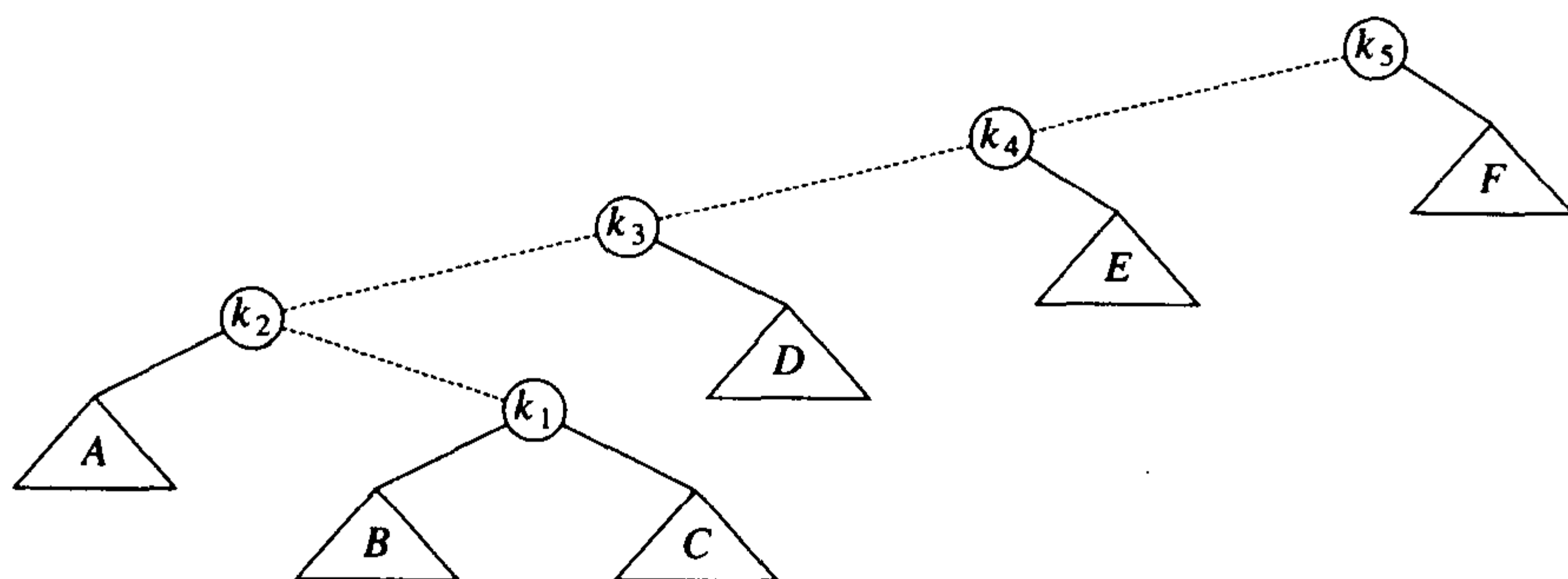
149



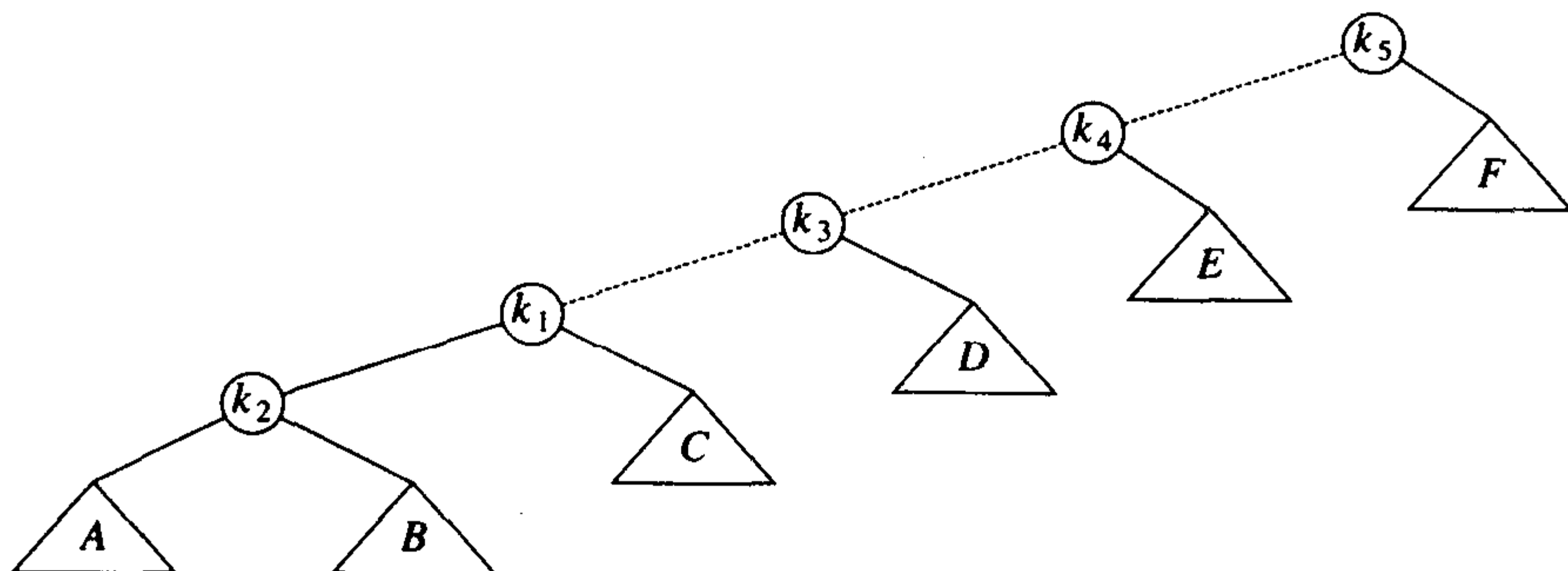
因为在许多应用中当一个结点被访问时，它就很可能不久再被访问。研究表明，这种情况的发生比人们预料的要频繁得多。另外，伸展树还不要求保留高度或平衡信息，因此它可以在某种程度上节省空间并简化代码（特别是当实现例程经过慎重考虑而写出的时候）。

#### 4.5.1 一个简单的想法（不能直接使用）

实施上面描述的重新构造的一种方法是执行单旋转，自底向上进行。这意味着将对访问路径上的每一个结点和它们的父结点实施旋转。作为例子，考虑在下面的树中对 $k_1$ 进行一次访问（一次find）之后所发生的情况。

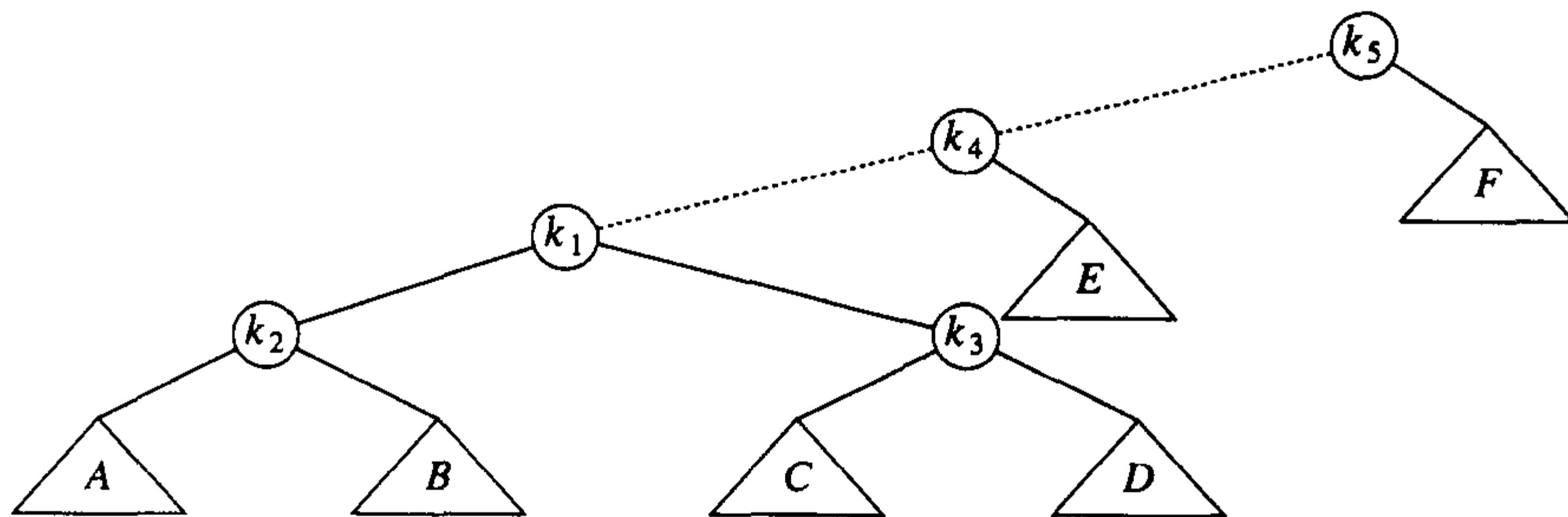


虚线是访问的路径。首先，在 $k_1$ 和它的父结点之间实施一次单旋转，得到下面的树。

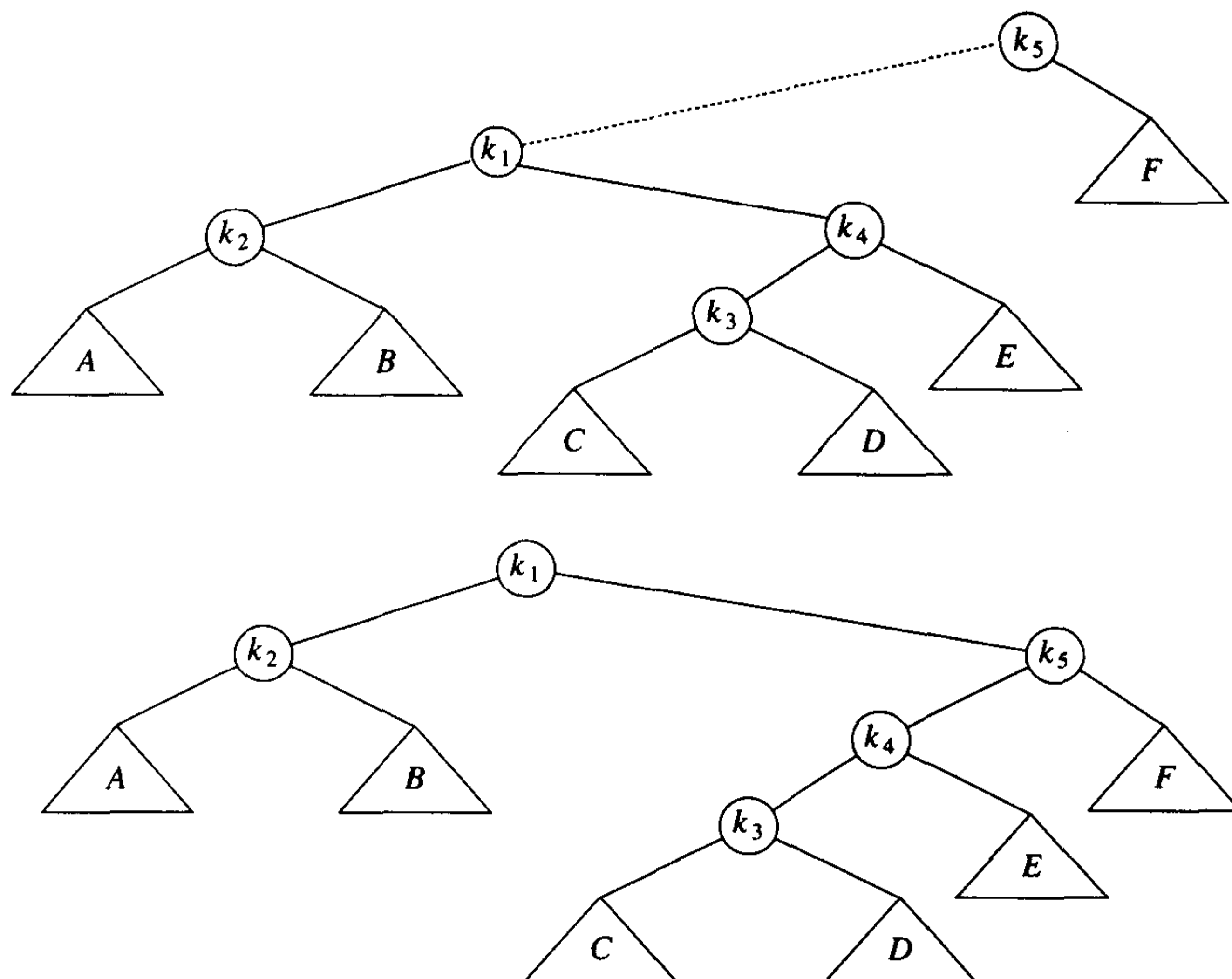


150

然后，在 $k_1$ 和 $k_3$ 之间旋转，得到下一棵树。



再实行两次旋转直到 $k_1$ 到达树根。



这些旋转的效果是将 $k_1$ 一直推向树根，使得对 $k_1$ 的进一步访问很容易（暂时的）。不足的是它把另外一个结点（ $k_3$ ）几乎推向和 $k_1$ 以前同样的深度。而对那个结点的访问又将把另外的结点向深处推进，如此等等。虽然这个策略使得对 $k_1$ 的访问花费时间减少，但是它并没有明显地改变（原先）访问路径上其他结点的状况。事实上可以证明，使用这种策略将会存在一系列 $M$ 个操作共需要 $\Omega(M \cdot N)$ 的时间，因此这个想法还不够好。证明这件事最简单的方法是考虑向初始的空树插入项 $1, 2, 3, \dots, N$ 所形成的树（请将这个例子算出）。由此得到一棵树，这棵树只由一些左儿子构成。由于建立这棵树总共花费时间为 $O(N)$ ，因此这未必不好。问题在于访问项为1的结点花费 $N-1$ 个单元的时间。在这些旋转完成以后，对项为2的结点的一次访问花费 $N-2$ 个单元的时间。依序访问所有项的总时间是 $\sum_{i=1}^{N-1} i = \Omega(N^2)$ 。在它们都被访问以后，该树转变回原始状态，而且我们可能重复这个访问顺序。

[151]

### 4.5.2 伸展

伸展（splaying）的方法类似于上面介绍的旋转的想法，不过在旋转如何实施上稍微有些选择的余地。我们仍然从底向上沿着访问路径旋转。令 $X$ 是在访问路径上的一个（非根）结点，我们将在这个路径上实施旋转操作。如果 $X$ 的父结点是树根，那么只要旋转 $X$ 和树根。这就是沿着访问路径上的最后的旋转。否则， $X$ 就有父亲（ $P$ ）和祖父（ $G$ ），存在两种情况以及对称的情形要考虑。第一种情况是之字形（zig-zag）情形（见图4-47）。这里， $X$ 是右儿子， $P$ 是左儿子（反之亦然）。如果是这种情况，那么我们执行一次就像AVL双旋转那样的双旋转。否则，出现另一种一字形（zig-zig）情形： $X$ 和 $P$ 或者都是左儿子，或者都是右儿子。在这种情况下，我们把图4-48左边的树变换成右边的树。

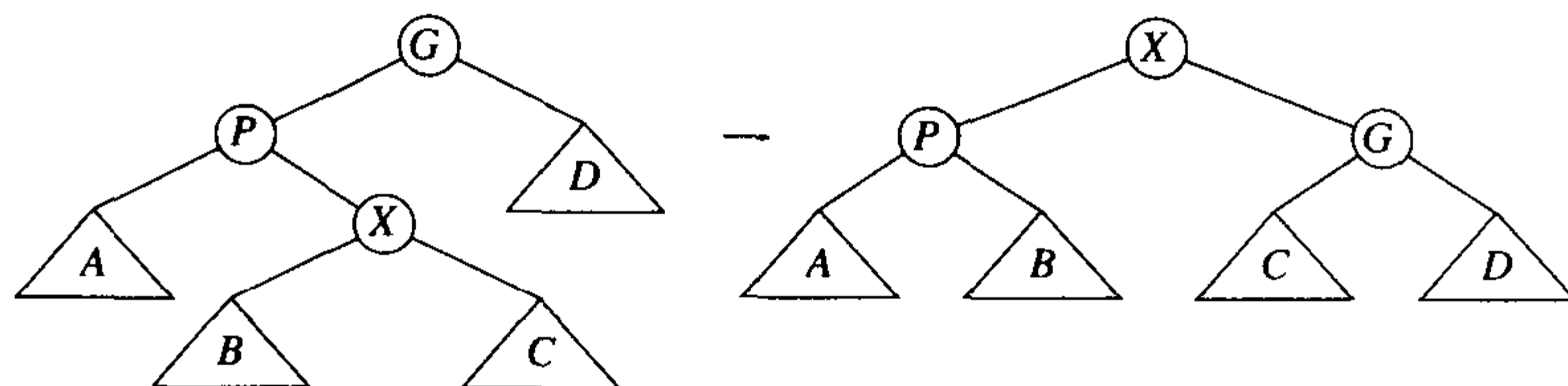


图4-47 之字形情形

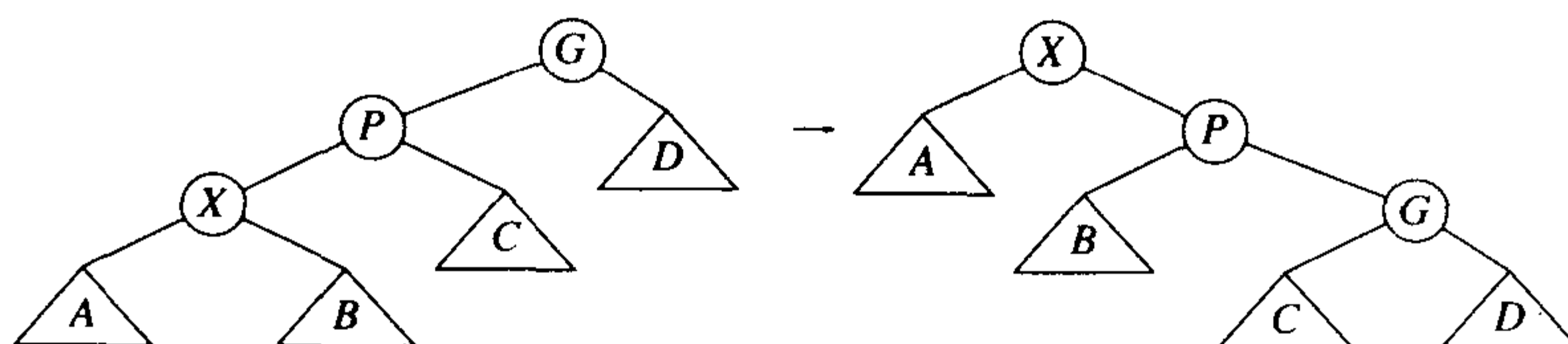
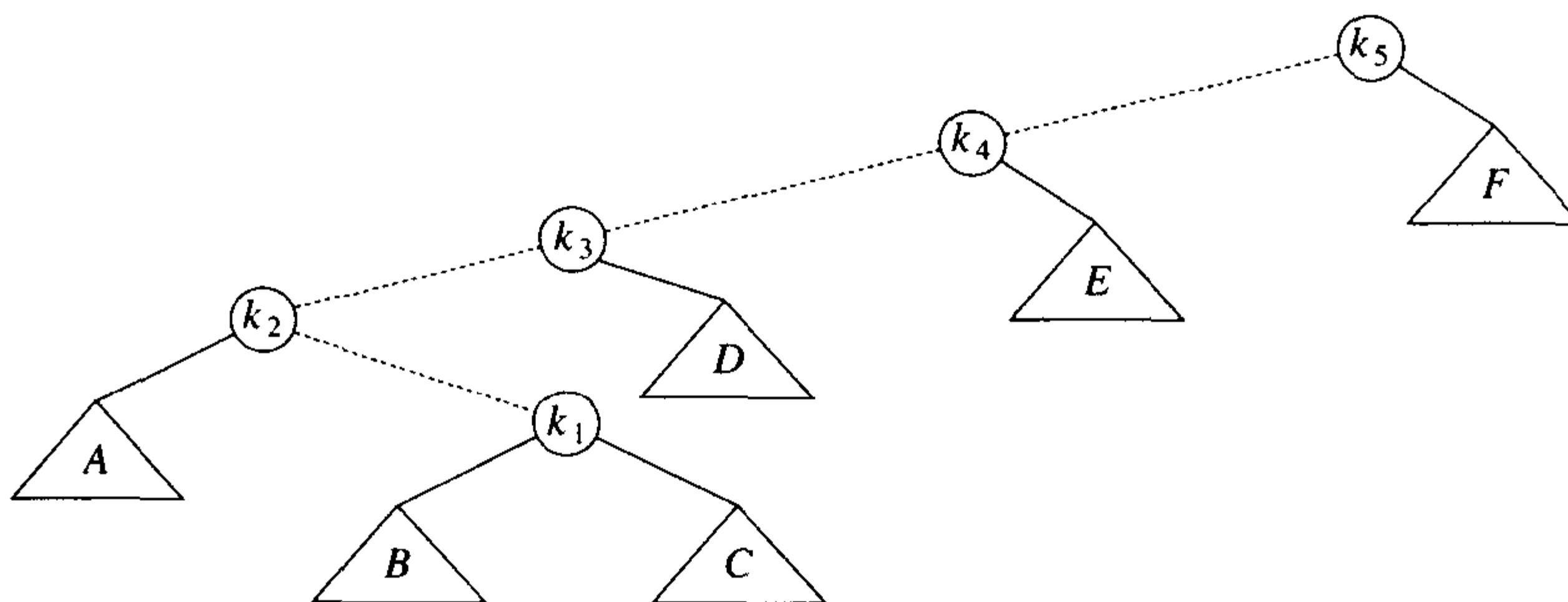
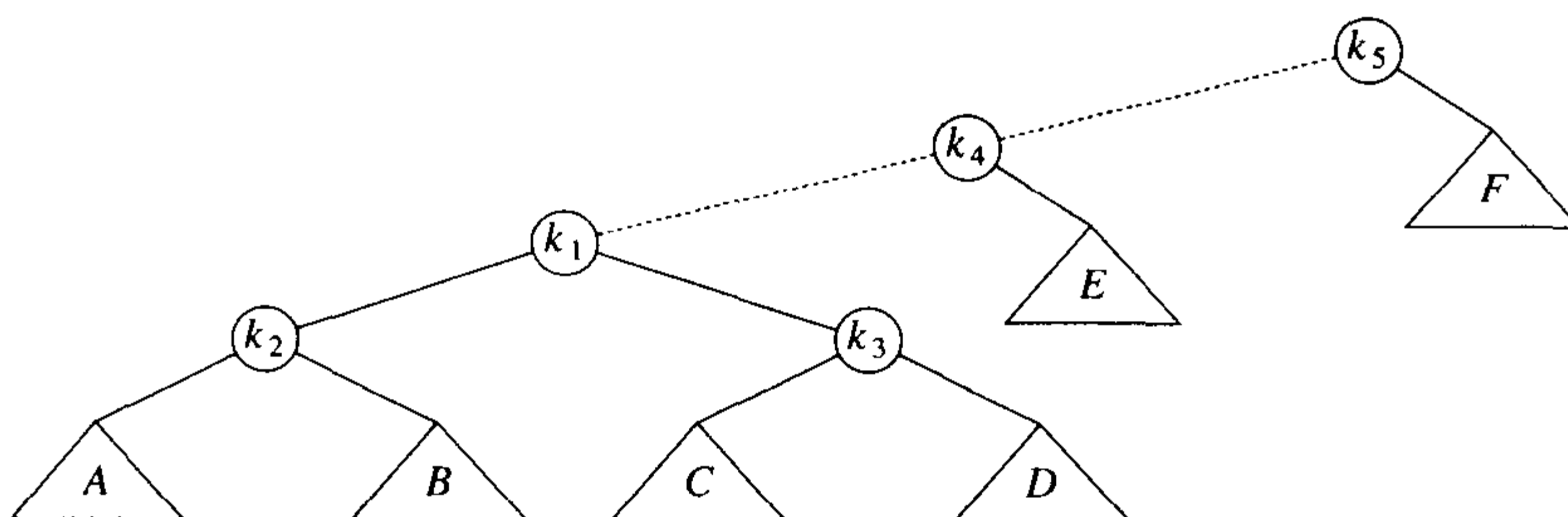


图4-48 一字形情形

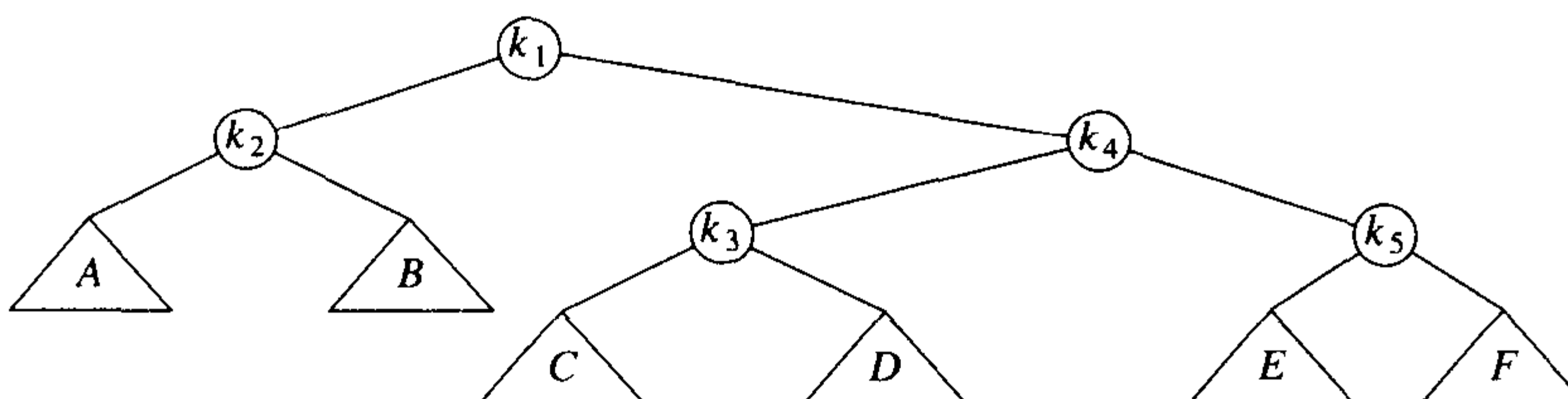
152 作为例子，考虑来自最后的例子中的树，对 $k_1$ 执行一次contains:



伸展的第一步是在 $k_1$ ，显然是一个之字形，因此我们用 $k_1$ 、 $k_2$ 和 $k_3$ 执行一次标准的AVL双旋转。得到如下的树。



在 $k_1$ 的下一步伸展是一个一字形，因此我们用 $k_1$ 、 $k_4$ 和 $k_5$ 做一字形旋转，得到最后的树。



虽然从一些小例子很难看出来，但是伸展操作不仅将访问的结点移动到根处，而且还有把访问路径上的大部分结点的深度大致减少一半的效果（某些浅的结点最多向下推两个层次）。

为了看出伸展与简单旋转的差别，再来考虑将1, 2, 3, ..., N各项插入到初始空树中去的效果。如前述可知共花费 $O(N)$ 时间并产生与一些简单旋转结果相同的树。图4-49指出在项为1的结点进行伸展的结果。区别在于，在对项为1的结点访问（花费 $N - 1$ 个单元的时间）之后，对项为2的结点的访问只花费 $N/2$ 个时间单位而不是 $N - 2$ 个时间单位；不存在像以前那么深的结点。

153

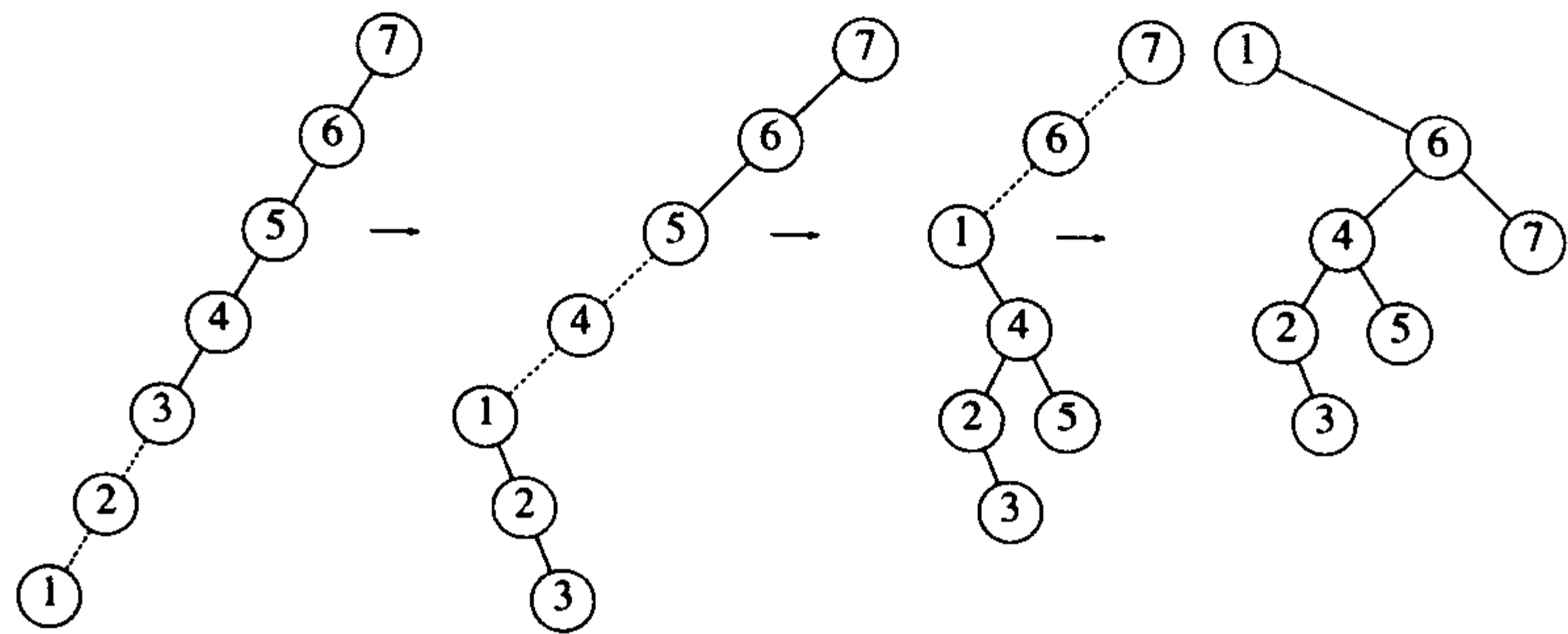


图4-49 在结点1伸展的结果

对项为2的结点的访问将把这些结点带到距根 $N/4$ 的深度之内，并且如此进行下去直到深度大约为 $\log N$ （ $N = 7$ 的例子太小，不能很好地看清这种效果）。图4-50至图4-58显示在32个结点的树中访问项1到9的结果，这棵树最初只含有左儿子。伸展树中没有简单旋转策略中常见的低效率的不良现象（实际上，这个例子只是一种非常好的情况。有一个相当复杂的证明指出，对于这个例子， $N$ 次访问共耗费 $O(N)$ 的时间）。

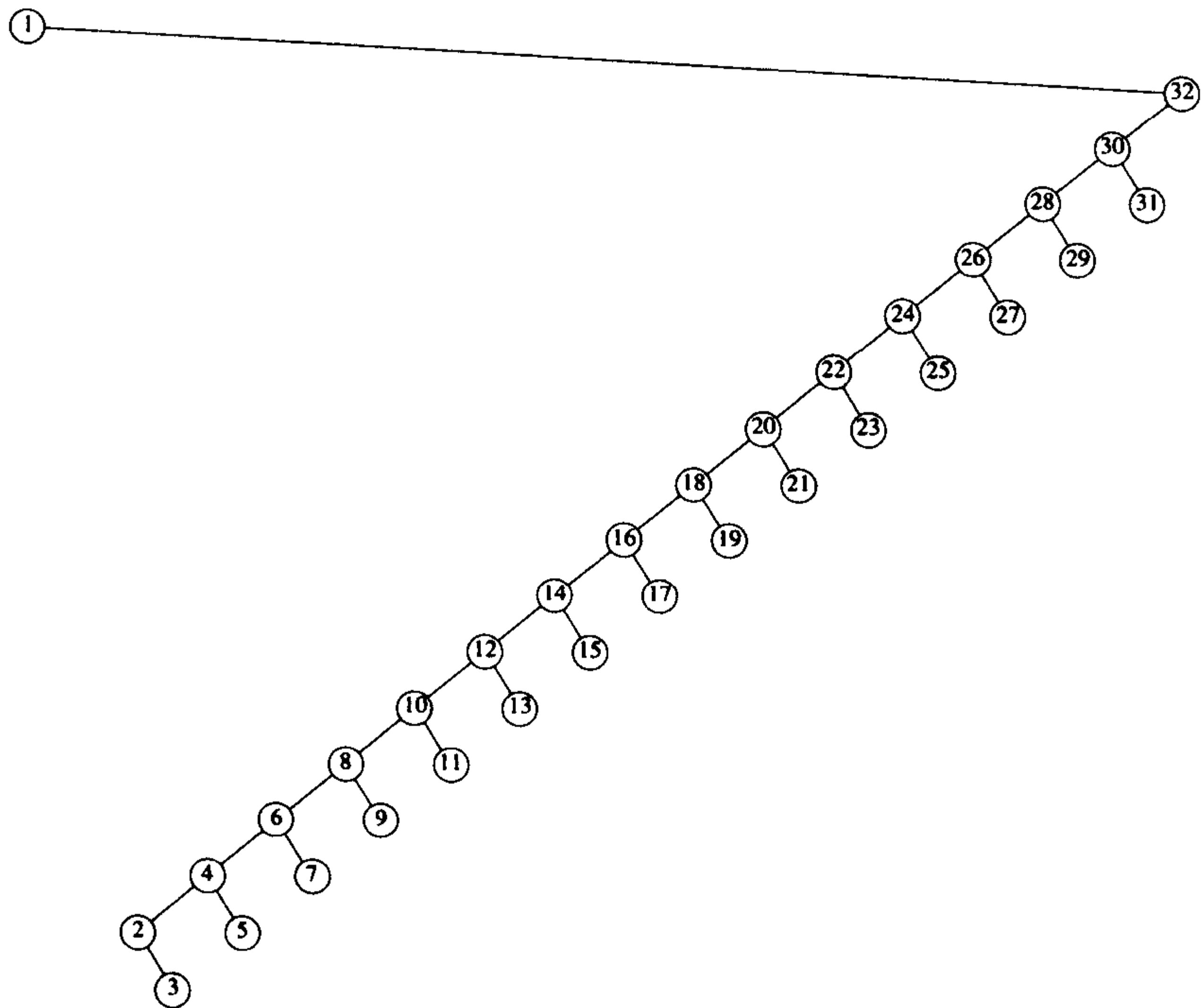


图4-50 将全部由左儿子构成的树在结点1伸展的结果



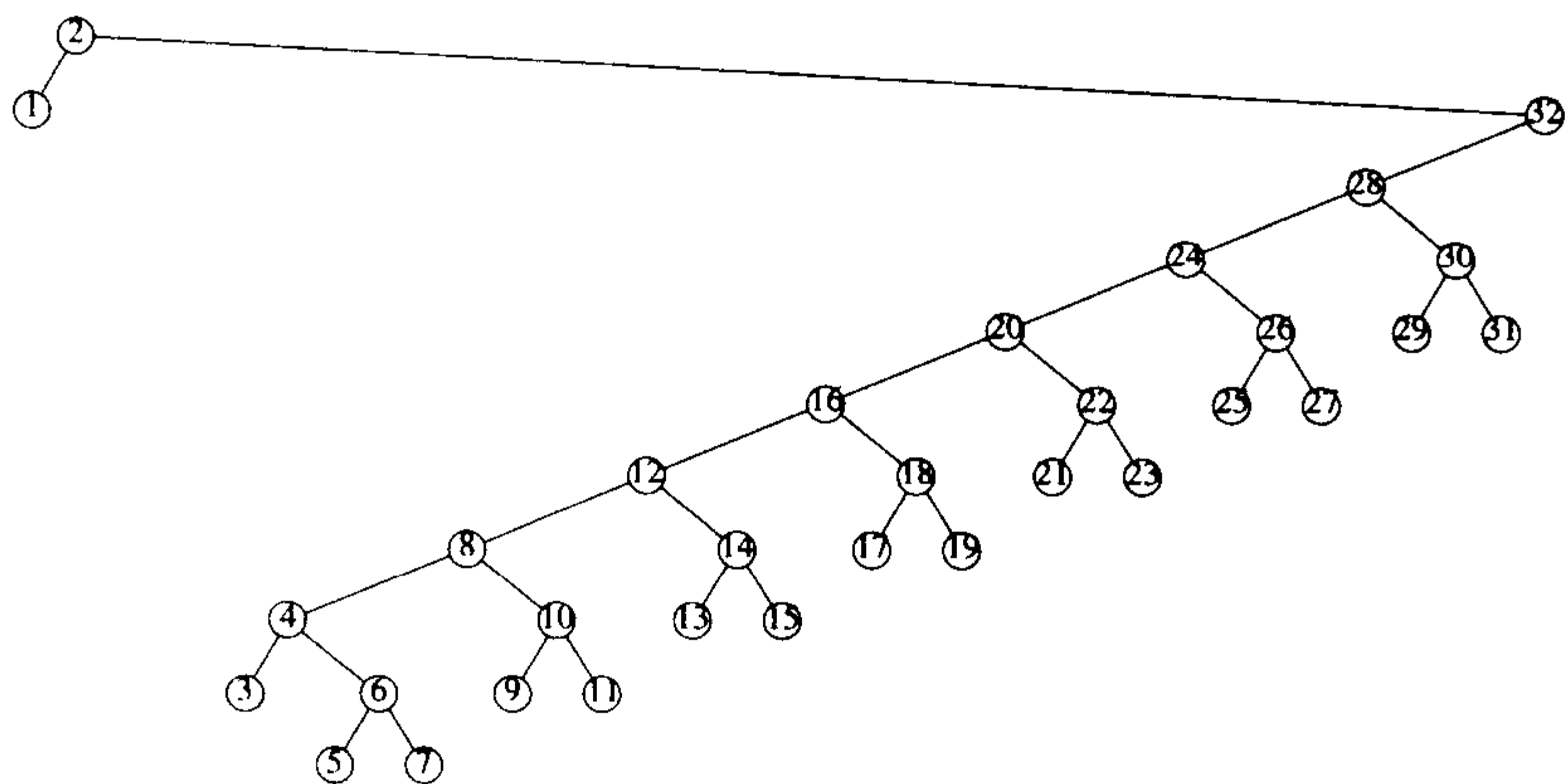


图4-51 将前面的树在结点2伸展的结果

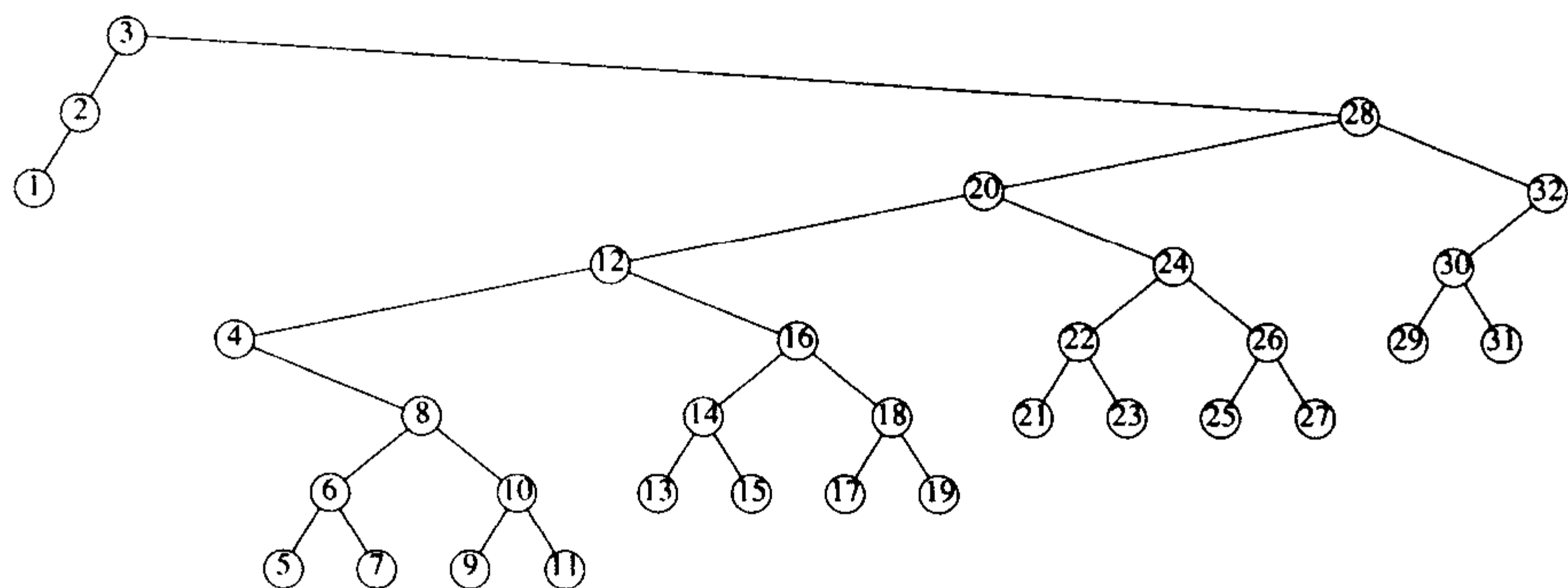


图4-52 将前面的树在结点3伸展的结果

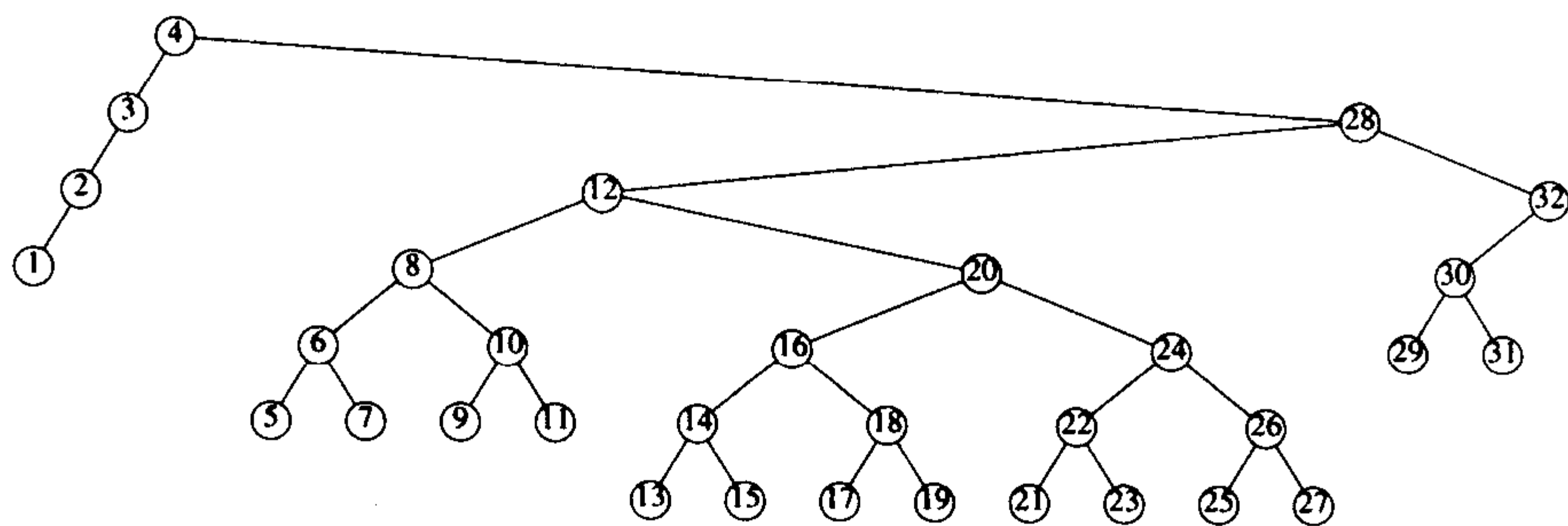


图4-53 将前面的树在结点4伸展的结果

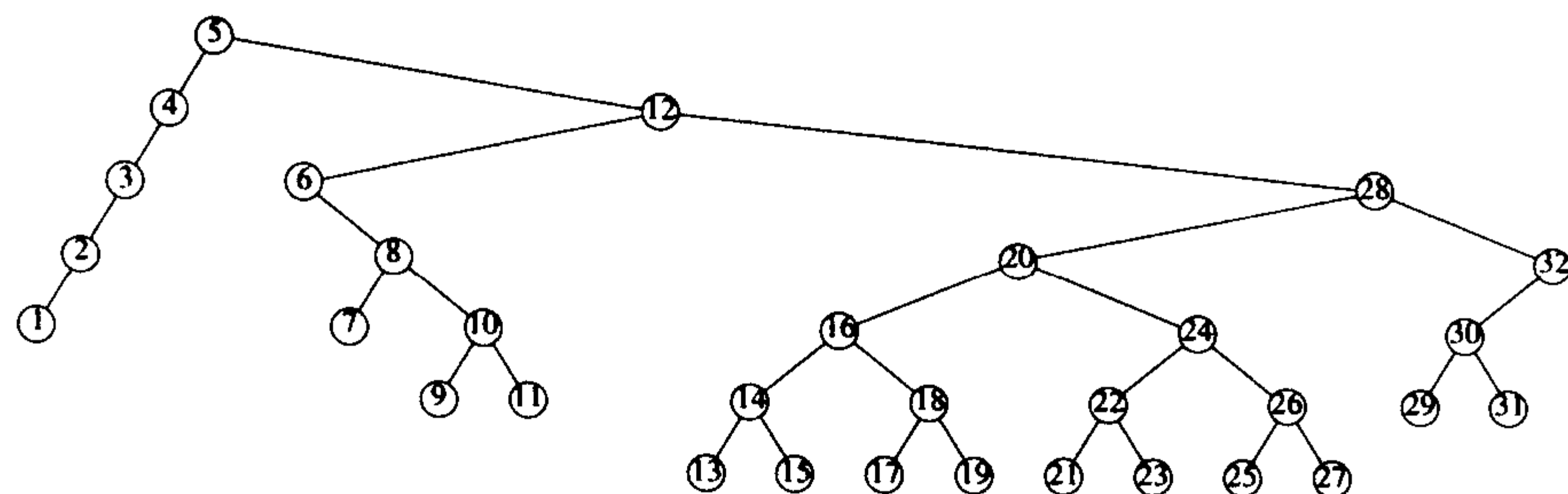


图4-54 将前面的树在结点5伸展的结果

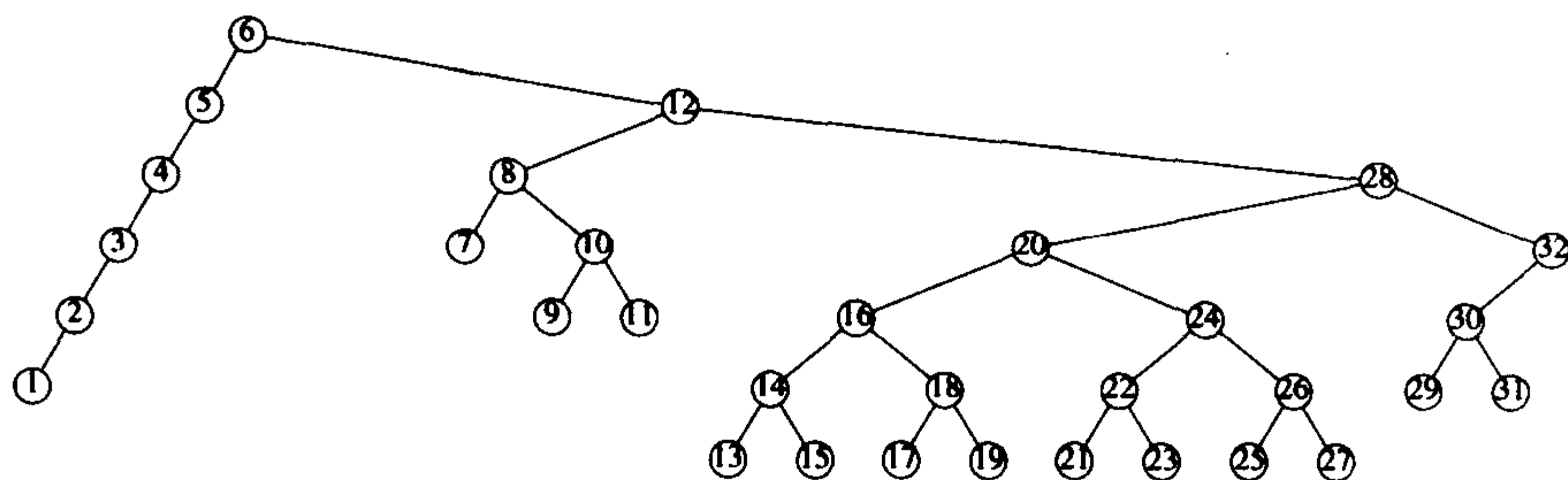


图4-55 将前面的树在结点6伸展的结果

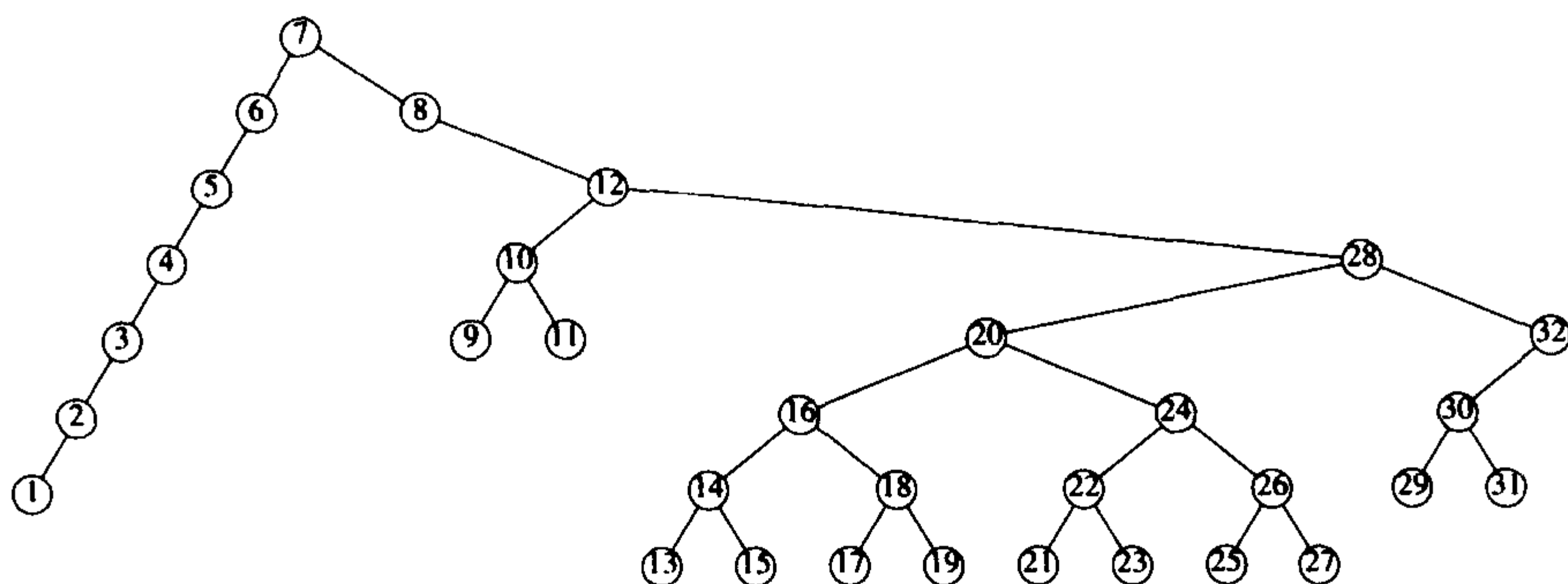


图4-56 将前面的树在结点7伸展的结果

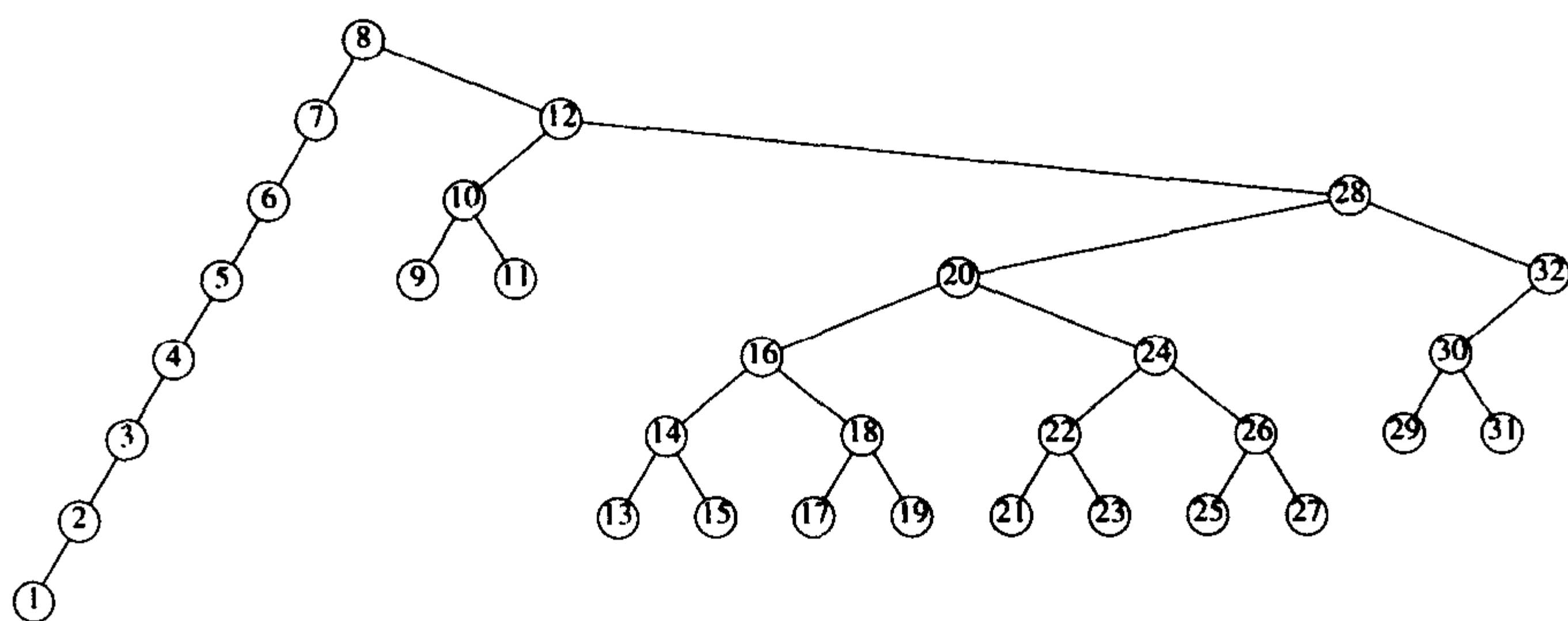


图4-57 将前面的树在结点8伸展的结果

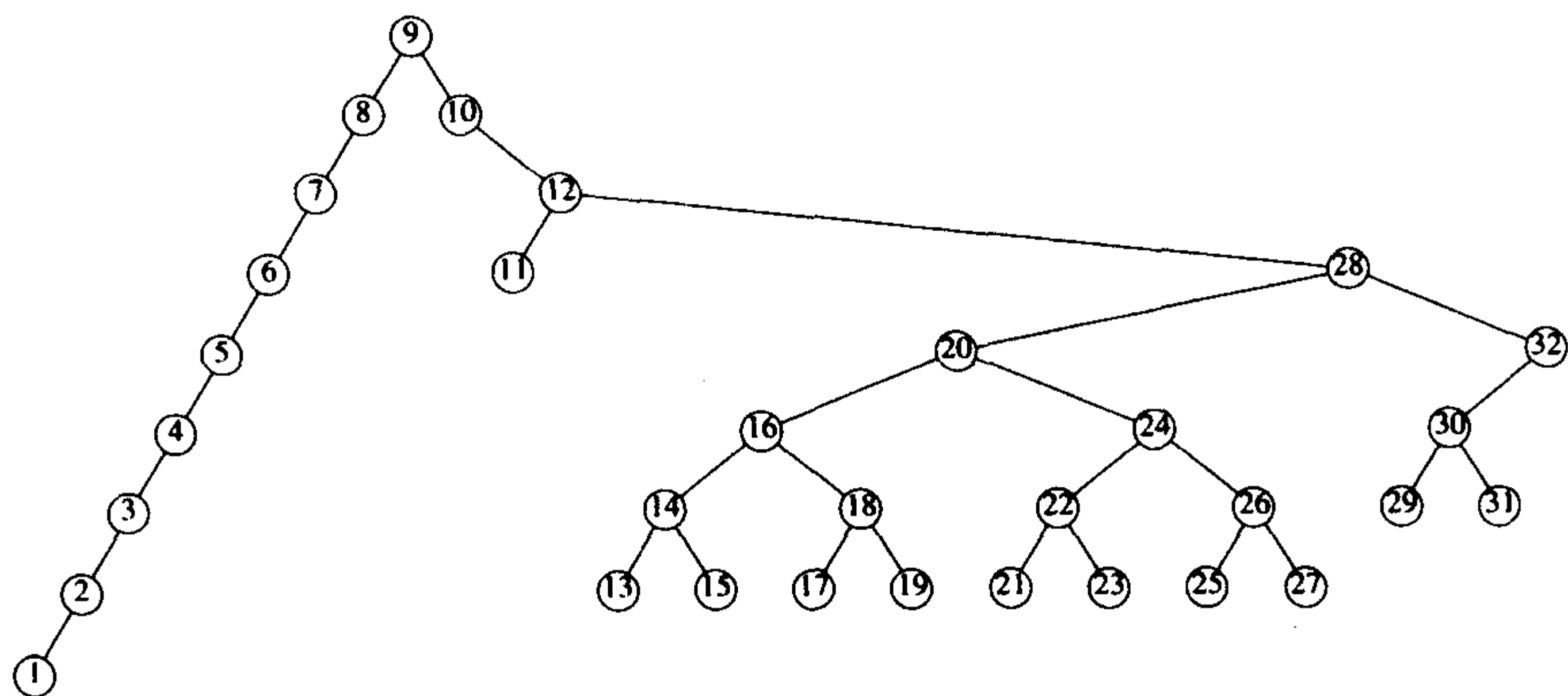


图4-58 将前面的树在结点9伸展的结果

这些图着重强调了伸展树基本的和关键的性质。当访问路径长而导致超出正常查找时间的时候，这些旋转将对未来的操作有益。当访问耗时很少的时候，这些旋转则不那么有益，甚至有害。极端的情形是经过若干插入而形成的初始树。所有的插入都是导致不好的初始树花费常数时间的操作。此时，我们会得到一棵很差的树，但是运行却比预计的快，从而总的运行时间较少，补偿了损失。这样，少数真正麻烦的访问却留给我们一棵几乎是平衡的树，其代价是我们必须付出某些已经省下的时间。在第11章将指出，每个操作的平均时间决不会大于 $O(\log N)$ ：即使偶尔有些不良操作，我们也总会获得这样的性能。

可以通过访问要被删除的结点实行删除的操作。这种操作将结点上推到根处。如果删除该结点，则得到两棵子树 $T_L$ 和 $T_R$ （左子树和右子树）。如果我们找到 $T_L$ 中的最大的元素（这很容易），那么这个元素就被旋转到 $T_L$ 的根下，而此时 $T_L$ 将有一个没有右儿子的根。我们可以使 $T_R$ 为右儿子从而结束删除。

对伸展树的分析很困难，因为树的结构经常变化。另一方面，伸展树的编程要比AVL树简单得多，这是因为要考虑的情形少并且没有平衡信息需要保留。一些实际经验指出，在实践中它可以转化成更快的程序代码，不过这种状况离完善还很远。最后，伸展树有几种变体，它们在实践中甚至运行得更好。有一种变体在第12章中将被完全编程实现。

## 4.6 树的遍历

由于二叉查找树中对信息已进行排序，因而按照排序的顺序列出所有的项很简单，图4-59中的递归方法进行的的就是这项工作。

```

1  /**
2   * Print the tree contents in sorted order.
3   */
4  void printTree( ostream & out = cout ) const
5  {
6      if( isEmpty( ) )
7          out << "Empty tree" << endl;
8      else
9          printTree( root, out );
10 }
11
12 /**
13  * Internal method to print a subtree rooted at t in sorted order.
14  */
15 void printTree( BinaryNode *t, ostream & out ) const
16 {
17     if( t != NULL )
18     {
19         printTree( t->left, out );
20         out << t->element << endl;
21         printTree( t->right, out );
22     }
23 }

```

图4-59 按顺序打印二叉查找树的例程

毫无疑问，该方法能够解决将项排序列出的问题。正如前面看到的，这类例程当用于树的时候则称为**中序遍历**（inorder traversal）（由于它依序列出了各项，因此是有意义的）。中序遍历的一般方法是首先处理左子树，然后是当前的结点，最后处理右子树。这个算法的有趣之处除它简

单的特性外，还在于其总的运行时间是 $O(N)$ 。这是因为在树的每一个结点处进行的工作是常数时间的。每一个结点访问一次，而在每一个结点进行的工作是检测是否NULL，建立两个函数调用并执行一个输出语句。由于在每个结点的工作花费常数时间以及总共有 $N$ 个结点，因此运行时间为 $O(N)$ 。

有时需要先处理两个子树然后才能处理当前结点。例如，为了计算一个结点的高度，首先需要知道它的子树的高度。图4-60中的程序就是计算高度的。由于检查一些特殊的情况总是有益的——当涉及递归时尤其重要，因此要注意这个例程声明树叶的高度为零，这是正确的。这种一般的遍历顺序称为**后序遍历**（postorder traversal），我们在前面也见到过。因为在每个结点的工作花费常数时间，所以总的运行时间也是 $O(N)$ 。

```

1  /**
2   * Internal method to compute the height of a subtree rooted at t.
3   */
4  int height( BinaryNode *t )
5  {
6      if( t == NULL )
7          return -1;
8      else
9          return 1 + max( height( t->left ), height( t->right ) );
10 }
```

图4-60 使用后序遍历计算树的高度的例程

第三种常用的遍历格式为**前序遍历**（preorder traversal）。这里，当前结点在其儿子结点之前处理。这种遍历可以用来利用结点深度标志每一个结点。

所有这些例程有一个共有的思想，那就是首先处理NULL的情形，然后才是其余的工作。注意，此处缺少一些附加的变量。这些例程仅仅传递对作为子树的根结点的引用，并没有声明或是传递任何附加的变量。程序越紧凑，一些愚蠢的错误出现的可能就越少。第四种遍历用得很少，叫作**层序遍历**（level-order traversal），我们以前尚未见到过。在层序遍历中，所有深度为 $d$ 的结点要在深度为 $d+1$ 的结点之前进行处理。层序遍历与其他类型的遍历不同的地方在于，它不是递归地实施的；它用到队列，而不使用递归所默认的栈。

## 4.7 B树

迄今为止，我们始终假设可以把整个数据结构存储到计算机的主存中。可是，如果数据太多主存装不下，那么就意味着必须把数据结构放到磁盘上。此时，因为大O模型不再适用，所以导致规则发生了变化。

问题在于，大O分析假设所有的操作都是相等的。然而，现在这么假设就不合适了，特别是涉及磁盘I/O的时候。例如，一台500 MIPS的机器每秒执行5亿条指令。这是相当快的，主要是因为速度主要依赖于电的特性。另一方面，磁盘是机械运动的，它的速度主要依赖于转动磁盘和移动磁头的时间。许多磁盘以7200 RPM旋转，也就是说，它7200转/min；因此，1转占用 $1/120$  s，即8.3 ms。平均认为磁盘转到一半的时候发现要寻找的信息；因此如果忽略其他因素，那么可以得到访问时间为8.3 ms（这是非常宽松的估计；9~11 ms的访问时间更为常见）。因此，每秒大约可以进行120次磁盘访问。若不和处理器的速度比较，那么这听起来还是相当不错的。可是考虑到处理器的速度，5亿条指令却花费相当于120次磁盘访问的时间。换句话说，一次磁盘访问的价值大约是400万条指令。当然，这里每一个数据都是粗略的计算，不过相对速度还是相当清楚的：



磁盘访问的代价太高了。不仅如此，处理器的速度还在以比磁盘速度快得多的速度增长（增长相当快的是磁盘的大小）。因此，为了节省一次磁盘访问，我们愿意进行大量的计算。几乎在所有的情况下，控制运行时间的都是磁盘访问的次数。于是，如果我们把磁盘访问次数减少一半，那么运行时间也将减少一半。

在磁盘上，典型的查找树执行如下：设我们想要访问佛罗里达州公民的驾驶记录。假设有1000万项，每一个键是32个字节（代表一个名字），而一个记录是256个字节。假设这些数据不能都装入主存，我们是在系统中20个用户之一（因此我们有1/20的资源）。这样，在1 s内，可以执行100万次指令，或者执行6次磁盘访问。

不平衡的二叉查找树是非常糟糕的。在最坏情形下它有线性的深度，从而可能需要1000万次磁盘访问。平均来看，一次成功的查找可能需要 $1.38 \log N$ 次磁盘访问，由于 $\log 10\,000\,000 \approx 24$ ，因此平均一次查找需要32次磁盘访问或5 s的时间。在一棵典型的随机构造的树中，我们预料会有一些结点比平均深度深3倍；它们需要大约100次磁盘访问或16 s的时间。AVL树多少要好一些。 $1.44 \log N$ 的最坏情形不可能发生，典型的情形是非常接近于 $\log N$ 。这样，一棵AVL树将平均使用大约25次磁盘访问，需要的时间是4 s。

我们想要把磁盘访问次数减小到一个非常小的常数，比如3或4；而且我们愿意写一个复杂的程序来做这件事，因为只要不是太冗长，机器指令基本上是不占时间的。由于典型的AVL树接近最优的高度，因此应该清楚的是，二叉查找树是不可行的。使用二叉查找树不能达到 $\log N$ 以下。直觉上看，解法是简单的：如果我们有更多的分支，那么我们就有更少的高度。这样，31个结点的理想二叉树有5层，而31个结点的5叉树则只有3层，如图4-61所示。一棵M叉查找树(M-ary search tree)可以有M路分支。随着分支增加，树的深度在减少。一棵完全二叉树的高度大约为 $\log_2 N$ ，而一棵完全M叉树的高度大约是 $\log_M N$ 。

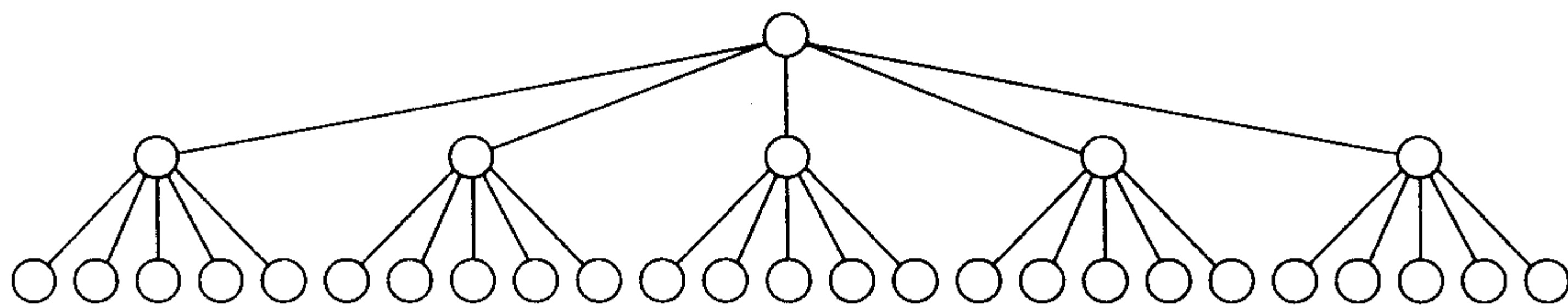


图4-61 31个结点的5叉树只有3层

可以以与建立二叉查找树大致相同的方式建立M叉查找树。在二叉查找树中，我们需要一个键来决定到底取用两个分支中的哪个；而在M叉查找树中需要M-1个键来决定选取哪个分支。为使这种方案在最坏的情形下有效，需要保证M叉查找树以某种方式得到平衡。否则，像二叉查找树，它可能退化成一个链表。实际上，我们甚至想要更加限制性的平衡条件。也就是说，我们不想让M叉查找树退化到甚至是二叉查找树，因为那样又将无法摆脱 $\log N$ 次访问了。

实现这种想法的一种方法是使用B树。这里描述基本的B树。<sup>1</sup>它有许多变种和改进，但实现起来多少要复杂些，因为有相当多的情形需要考虑。不过，容易看到，原则上B树保证只有少数的磁盘访问。

阶为M的B树是一棵具有下列结构特性的树<sup>2</sup>：

- (1) 数据项存储在树叶上。

1. 这里所描述的是通常称为B<sup>+</sup>树的树。

2. 特性(3)和(5)对于前L次插入必须要放宽。

- (2) 非叶结点存储直到  $M-1$  个键，以指示搜索的方向；键  $i$  代表子树  $i+1$  中的最小的键。
- (3) 树的根或者是一片树叶，或者其儿子数在 2 和  $M$  之间。
- (4) 除根外，所有非树叶结点的儿子数在  $\lceil M/2 \rceil$  和  $M$  之间。
- (5) 所有的树叶都在相同的深度上并有  $\lceil L/2 \rceil$  和  $L$  之间个数据项，稍后描述  $L$  的确定。

图4-62显示了5阶B树的一个例子。注意，所有的非树叶结点的儿子数都在3和5之间（从而有2到4个键）；根可能只有两个儿子。这里， $L=5$ 。在这个例子中  $L$  和  $M$  恰好是相同的，但这不是必需的。由于  $L$  是 5，因此每片树叶有 3 到 5 个数据项。要求结点一半满，将保证B树不致退化成简单的二叉树。虽然存在改变该结构的各种B树的定义，但大部分在一些次要的细节上变化，这里的定义是一种流行的形式。

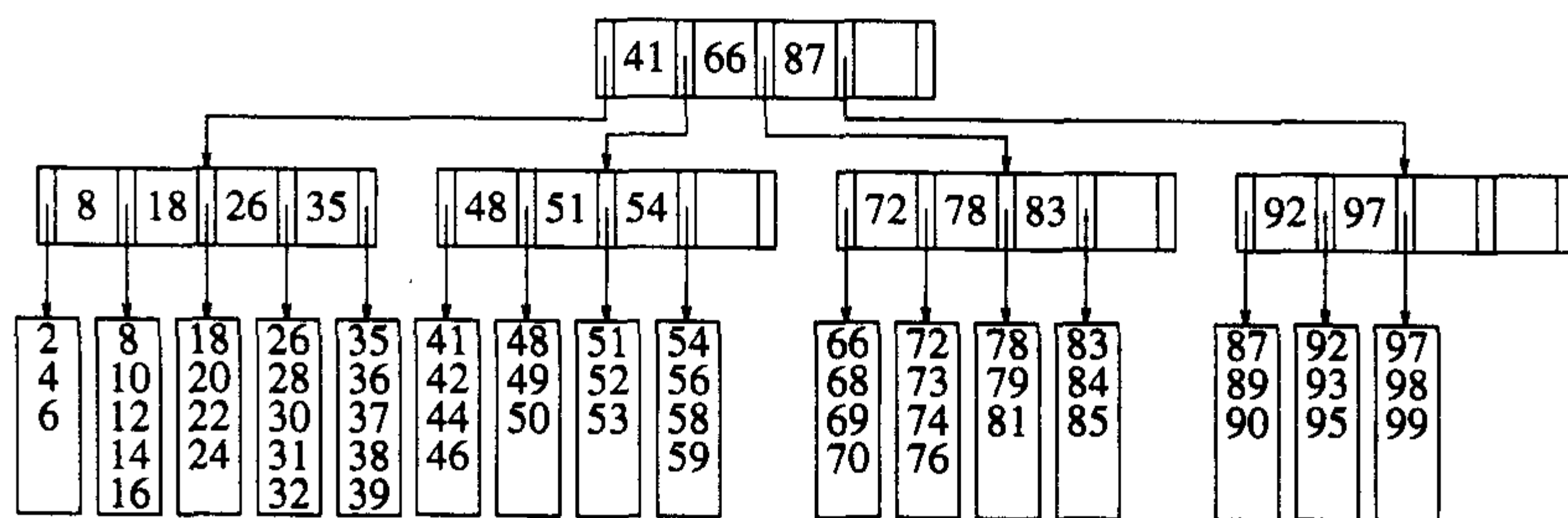


图4-62 5阶B树

每个结点代表一个磁盘区块，于是我们根据所存储的项的大小选择  $M$  和  $L$ 。例如，设一个区块容纳 8192 字节。在上面的佛罗里达的例子中，每个键使用 32 个字节。在一棵  $M$  阶 B 树中，我们有  $M-1$  个键，总数为  $32M-32$  字节，再加上  $M$  个分支。由于每个分支基本上都是另外一些磁盘区块，因此我们可以假设一个分支是 4 个字节。这样，这些分支共用  $4M$  个字节。一个非叶结点总的内存需求为  $36M-32$  个字节。使得不超过 8192 字节的  $M$  的最大值是 228。因此，我们选择  $M=228$ 。由于每个数据记录是 256 字节，因此我们能够把 32 个记录装入一个区块中。于是，我们选择  $L=32$ 。这样就保证每片树叶有 16 到 32 个数据记录以及每个内部结点（除根外）至少以 114 种方式分叉。

161

由于有 1000 万个记录，因此至多存在 625 000 片树叶。由此得知，在最坏情形下树叶将在第 4 层上。更具体地说，最坏情形的访问次数近似地由  $\log_{M/2} N$  给出，这个数可以有 1 的误差（例如，根和下一层可以存放在主存中，使得经过长时间运行后磁盘访问将只对第 3 层或更深层是需要的）。

剩下的问题是如何向 B 树添加项和从 B 树删除项的问题；下面将概述所涉及的原理。注意，许多主题以前见到过。

我们首先考察插入。设想要把 57 插入到图 4-62 的 B 树中。沿树向下查找揭示出它不在树中。此时把它作为第 5 个儿子添加到树叶中。注意可能要为此重新组织该树叶上的所有数据。然而，与磁盘访问相比（在这种情况下它还包含一次磁盘写），做这件事的开销是可以忽略的。

当然，这是相对简单的，因为该树叶还没有装满。设现在想要插入 55。图 4-63 显示了一个问题：55 想要插入其中的那片树叶已经满了。不过解法却不复杂：由于现在有  $L+1$  项，因此把它们分成两片树叶，这两片树叶保证都有所需要的记录的最小个数。我们形成两片树叶，每叶 3 项。写这两片树叶需要 2 次磁盘访问，更新它们的父结点需要第 3 次磁盘访问。注意，在父结点中键和分支均发生了变化，但是这种变化是以容易计算的受控方式处理的。最后得到的 B 树在图 4-64 中示出。虽然分裂结点是耗时的，因为它至少需要 2 次附加的磁盘写，但它相对很少发生。例如，如果  $L$  是 32，那么当结点被分裂时，将分别建立具有 16 和 17 项的两片树叶。对于有 17 项的树叶，

162

可以再执行15次插入而不用另外的分裂。换句话说，对于每次分裂，大致存在 $L/2$ 次非分裂的插入。

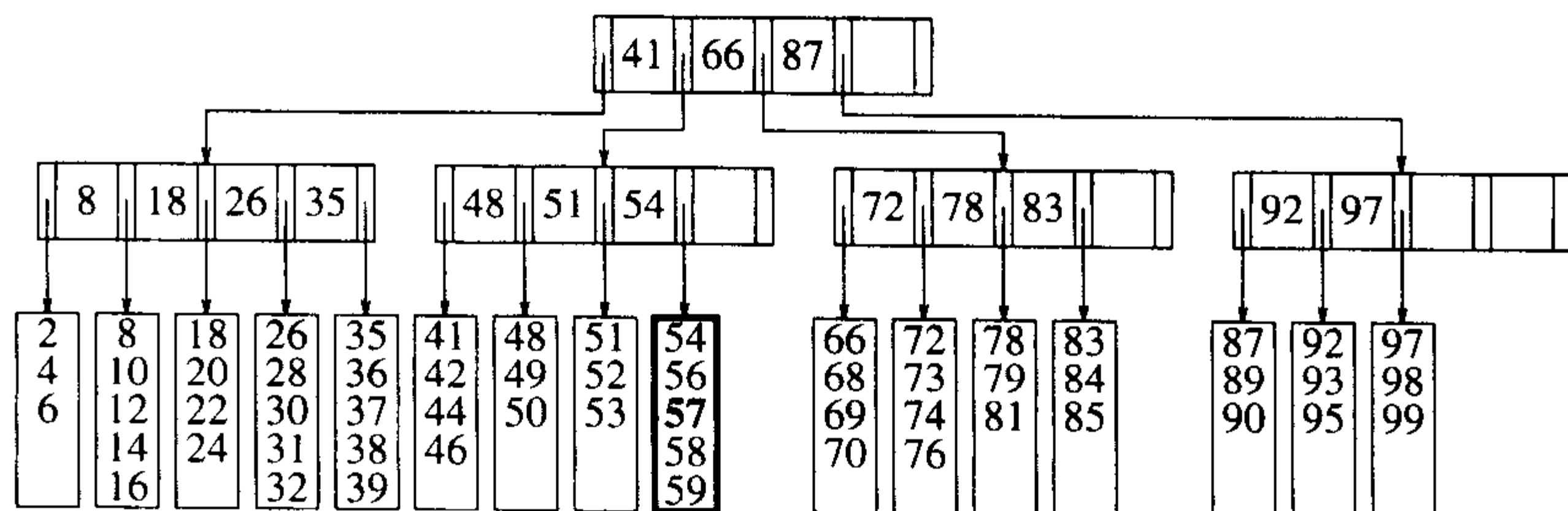


图4-63 将57插入到图4-62的树中后的B树

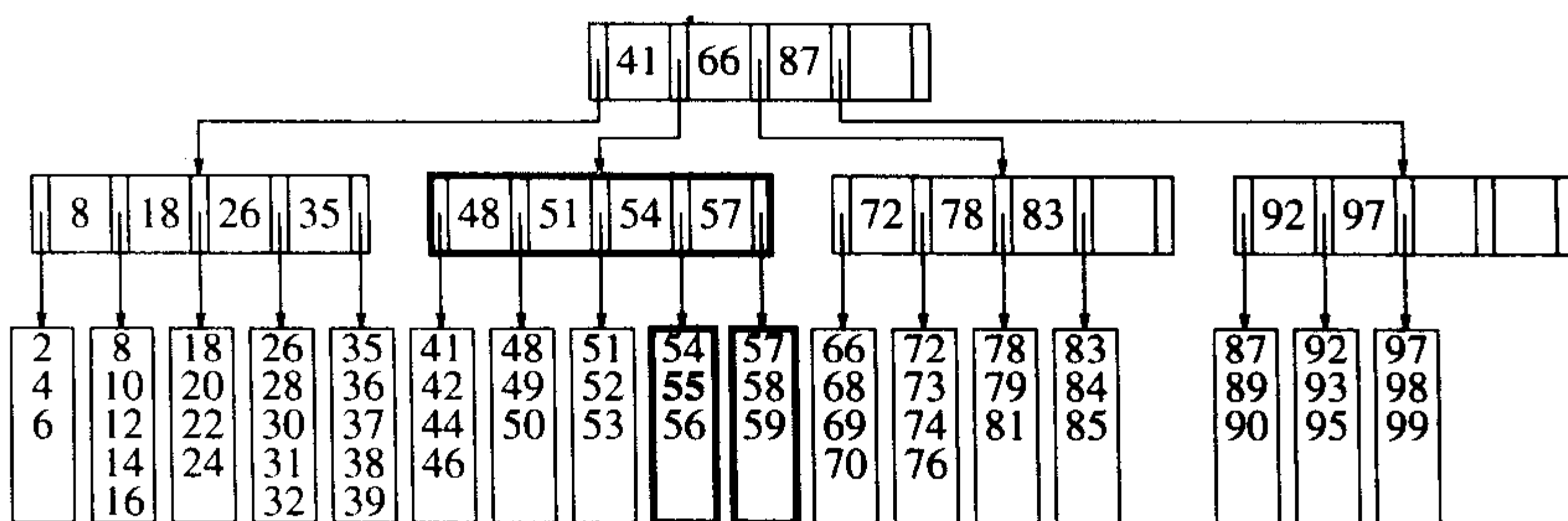


图4-64 将55插入到图4-63的B树中引起分裂成两片树叶

前面例子中的结点分裂之所以行得通，是因为其父结点的儿子个数尚未满员。如果满员了又会怎样呢？例如，假设想要把40插入到图4-64的B树中去。此时必须把包含键35到39而现在又要包含40的树叶分裂成2片树叶。但是这将使父结点有6个儿子，可是它只能有5个儿子。因此，要分裂这个父结点。结果在图4-65中给出。当父结点被分裂时，我们必须更新那些键以及还有父结点的父亲的值，这样就导致额外的两次磁盘写（从而这次插入需要5次磁盘写）。然而，虽然由于

163

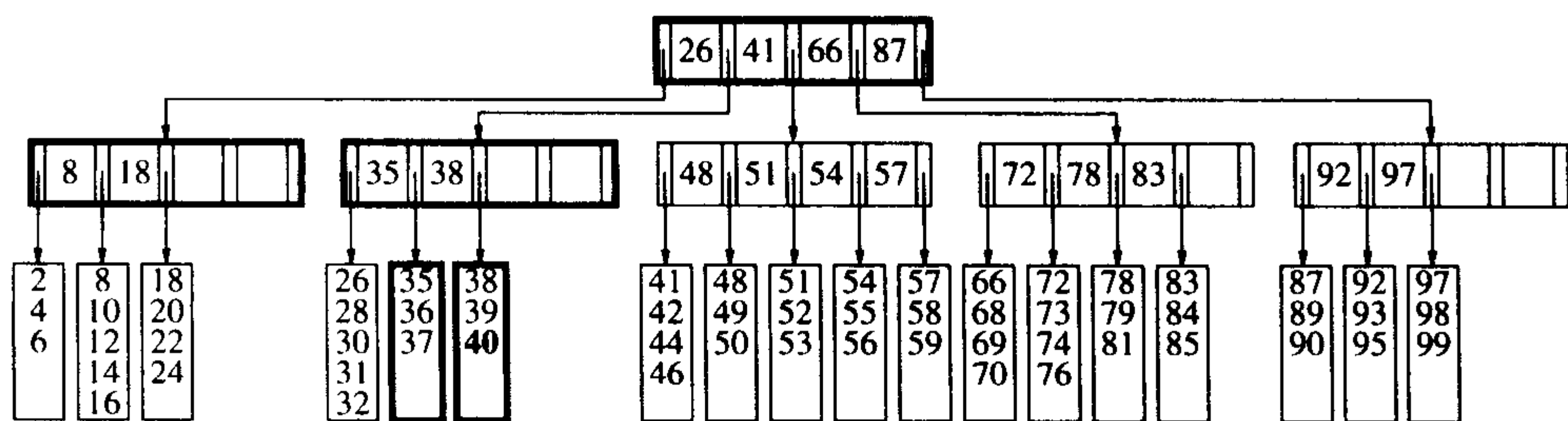


图4-65 把40插入到图4-64的B树中，引起树叶被分裂成两片然后又造成父结点的分裂

正如这里的情形所示，当一个非树叶结点分裂时，它的父结点得到了一个儿子。如果父结点的儿子个数已经达到规定的限度怎么办呢？在这种情况下，继续沿树向上分裂结点直到找到一个父结点它不需要再分裂，或者到达树根。如果分裂树根，那么就得到两个树根。显然这是不可接受的，但可以建立一个新的根，这个根以分裂得到的两个树根作为它的两个儿子。这就是准许树根可以最少有两个儿子的特权的原因。这也是B树增加高度的唯一的方式。毫无疑问，一路向上分裂直到根的情况是一种特别少见的例外事件，因为一棵具有4层的树意味着在整个插入序列中



已经被分裂了3次（假设没有删除发生）。事实上，任何非树叶结点的分裂也是相当少见的。

还有其他一些方法处理儿子过多的情况。一种方法是在相邻结点有空间时把一个儿子交给该邻结点领养。例如，为了把29插入到图4-65的B树中，可以把32移到下一片树叶而腾出一个空间。这种方法要求对父结点进行修改，因为键受到了影响。然而，它趋向于使得结点更满从而在长时间运行中节省空间。

可以通过查找要被删除的项并将其删除来执行删除操作。问题在于，如果被删的项所在的树叶的数据项数已经是最小值，那么删除后它的项数就低于最小值了。我们可以通过在邻结点本身没有达到最小值时领养一个邻项来矫正这种状况。如果邻结点也已经是最小值，那么可以与相邻结点联合以形成一片满叶。可是，这意味着其父结点失去一个儿子。如果失去儿子的结果又引起父结点的儿子数低于最小值，那么使用相同的策略继续进行。这个过程可以一直上行到根。根不可能只有一个儿子（要是允许根有一个儿子，那可就太愚蠢了）。如果这个领养过程的结果使得根只剩下一个儿子，那么删除该根并让它的这个儿子作为树的新根。这是B树降低高度的唯一的方式。例如，假设想要从图4-65的B树中删除99。由于那片树叶只有两项而它的邻结点也已经是最小值3了，因此把这些项合并成有5项的一片新的树叶。结果，它们的父结点只有两个儿子了。这时该父结点可以从它的邻结点领养，因为邻结点有4个儿子。领养的结果使得双方都有3个儿子，结果如图4-66所示。

164

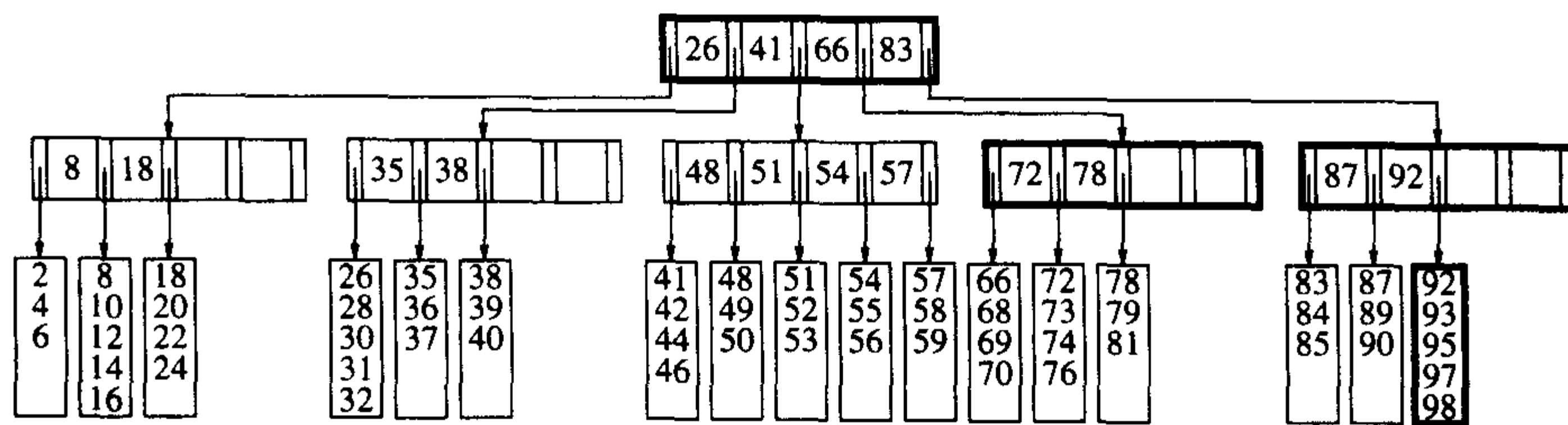


图4-66 从图4-65的B树中删除99后的B树

## 4.8 标准库中的set和map

在第3章中讨论了STL中的容器vector和list，这两者对于查找来说是不够用的。相应地，STL提供了两个附加的容器set和map，这两个容器保证了基本操作（如插入、删除和查找）的对数时间开销。

### 4.8.1 set

set是一个排序后的容器，该容器不允许重复。许多用于访问vector和list中的项的例程也适用于set。特别地，iterator和const\_iterator类型是嵌套于set的，该类型允许遍历set。vector和list的几个方法在set中有完全相同的名字，包括begin、end、size和empty。图3-6中的print函数模板在传递的参数为set时也可以工作。

set特有的操作是高效的插入、删除和执行基本查找。

插入例程被恰当地命名为insert。然而，因为set不允许重复，所以，对insert来说，有可能会出现插入失败的情况。因此，我们希望返回类型是一个可以指示这种情况的布尔变量。然而，insert返回的是一个比bool类型复杂得多的类型。这是因为insert也返回一个iterator来给出当insert返回时x的位置。这个iterator或者指向新插入的项，或者指向导致insert失败的已



有项。这个iterator是很有用的，因为，如果知道项的位置的话，就可以快捷地删除该项。可以直接获得包含该项的结点，从而避免了查找操作。

STL定义了一个名为pair的类模板，该类模板比struct多两个用来访问pair的两项的成员first和second。下面是两个不同的insert例程：

```
pair<iterator,bool> insert( const Object & x );
pair<iterator,bool> insert( iterator hint, const Object & x );
```

单参数insert的执行如上所示。双参数insert允许对 x将要插入的位置的线索说明。如果线索很精确，那么，插入就很快，通常为 $O(1)$ 。如果不精确的话，就需要用常规的插入算法来完成，此时的执行与单参数insert相同。例如，下面的代码中使用双参数insert就比使用单参数insert要快得多：

```
set<int> s;
for( int i=0; i<1000000; i++)
    s.insert(s.end(),i);
```

有几个版本的erase：

```
int erase( const Object & x);
iterator erase( iterator itr );
iterator erase( iterator start, iterator end);
```

165

第一个单参数erase删除x（如果找到的话），然后返回删除的元素个数。很明显，这个返回值不是0就是1。第二个单参数erase的执行与在vector和list中完全一样。删除由iterator指定的位置的对象，返回的iterator指向在调用erase之前紧跟在itr的下一个位置的元素，然后使itr失效，因为此时的itr已经没用了。双参数erase的执行与在vector或list中相同。删除从start开始、到end终止的所有的项（不包括end）。

对于查找，set提供一个优于返回布尔变量的contains例程的find例程，该例程返回一个iterator用以指向项的位置（如果查找失败就指向末端标志符）。这在不占用运行时间的前提下，提供了相当可观的更多的信息。find的形式如下：

```
iterator find( const Object & x ) const;
```

默认情况下，排序操作使用less<Object>函数对象实现，而该函数对象是通过对Object调用operator<来实现的。另一种可替代的排序方案可以通过具有函数对象类型的set模板来举例说明。例如，可以生成一个存储string对象的set，通过使用CaseInsensitiveCompare函数对象（其代码见图1-22）来忽略字符的大小写。在下面的代码中，set s的大小为1。

```
set<string,CaseInsensitiveCompare> s;
s.insert( "Hello" ); s.insert("HeLLo");
cout << "The size is: " << s.size() << endl;
```

#### 4.8.2 map

map用来存储排序后的由键和值组成的项的集合。键必须唯一，但是多个键可以对应同一个值。因此，值不需要唯一。在map中的键保持逻辑排序后的顺序。

map的执行类似于用pair例示的set。其中的比较函数仅仅涉及键<sup>1</sup>。因此，map支持begin、end、size和empty，但是基本的迭代器是一个键—值对。换句话说，对iterator itr, \*itr是pair<KeyType,ValueType>类型的。map也支持insert、find和erase。对于insert，必须提供pair<KeyType,ValueType>对象。虽然find仅需要一个键，返回的iterator还是指向一个

1. 类似于 set，一个可选的模板参数可以用来说明比较函数，该比较函数不同于 less<KeyType>。

pair。通常使用这些操作都是不值得的，因为这会导致昂贵的语法累赘。

幸运的是，map有一个重要的额外操作可以获得简单的语法。如下所示是对map的数组索引操作符重载：

```
ValueType & operator[] ( const KeyType & key );
```

operator[]的语法如下。如果在map中存在key，就返回指向相应的值的引用。如果在map中不存在key，就在map中插入一个默认的值，然后返回指向这个插入的默认值的引用。这个默认值通过应用零参数构造函数获得，如果是基本类型的话就是0。这些语法不允许修改函数版本的operator[]，因此operator[]不能用于常量的map。例如，如果在例程中map是通过常量引用来传递的，那么，operator[]就不可用。

图4-67的代码段例举了两个访问map的项的技术。首先观察第3行，左边调用operator[]，因此插入"Pat"和一个值为0的double到map，同时返回指向这个double的引用。然后赋值将map中的double改为75 000。第4行输出75 000。遗憾的是，第5行插入"Jan"和工资"0.0"到map中，并打印出来。这或许能、或许不能得到正确的结果，这取决于应用程序。如果区分在map中的和不在map中的项很重要的话，或者，不插入到map中（因为不可修改），那么可以使用7~12行所示的一个替代的方法。那里有一个对find的调用。如果键没有找到，iterator就是末端标志符并且可以进行测试。如果键没有找到，我们可以访问在这个对中由iterator引用的第二项，该项为与键对应的值。如果itr是iterator、而不是const\_iterator的话，就可以进行赋值itr->second。

```
1 map<string,double> salaries;
2
3 salaries[ "Pat" ] = 75000.00;
4 cout << salaries[ "Pat" ] << endl;
5 cout << salaries[ "Jan" ] << endl;
6
7 map<string,double>::const_iterator itr;
8 itr = salaries.find( "Chris" );
9 if( itr == salaries.end( ) )
10     cout << "Not an employee of this company!" << endl;
11 else
12     cout << itr->second << endl;
```

图4-67 访问map中的值

### 4.8.3 set和map的实现

C++需要set和map支持在最坏情况下对基本的操作insert、erase和find仅消耗对数时间。相应的，底层实现是平衡二叉查找树。典型的用法不是使用AVL树，而是常常使用自顶向下红黑树。自顶向下红黑树将在12.2节讨论。

在实现set和map时，一个重要的问题就是需要提供对迭代器类的支持。当然，在程序内部，迭代器在迭代过程中始终保持一个指针指向“当前”结点。困难的地方是如何高效地将迭代器推进至下一个结点。有几种可能的解决方案，其中的一部分例举如下：

(1) 当迭代器构造完成后，每一个迭代器都将一个包含set项的数组作为自己的数据存储。这没有用：这使得在修改过set后返回一个迭代器的任何例程的实现都不可能高效。例如一些版本的erase和insert。

(2) 使迭代器维持一个栈，用来存储通向当前结点的路径上的结点。基于这个信息，可以推出在迭代器中的下一个结点，它可能是当前结点的右子树所包含的最小项，或者是最近的在其左

166

167

子树中包含当前结点的祖先。这使得迭代器有一点大，并且使得迭代器的代码很笨拙。

(3) 使查找树中的每一个结点除了存储其儿子外，也存储其父亲。迭代器不会很大，但是现在每个结点都需要额外的存储空间，而且迭代代码还是很笨拙。

(4) 使每个结点保持额外的链接：一个至下一个较小结点，另一个至下一个较大结点。这样也占用空间，但是此时的迭代过程就很容易实现，而且很容易对这些链接进行维护。

(5) 仅为那些左侧或右侧的链接为NULL的结点保持额外的链接，通过使用额外的布尔变量使得例程可以指示出是否一个左链接正在作为标准二叉查找树的左链接或者至下一个较小结点的链接而使用，对于右结点也做同样处理（练习4.49）。这种思想称为**线索树**（threaded tree），在许多STL的实现中都有用到。

#### 4.8.4 使用几个map的例子

许多词都和其他词相似。例如，替换第一个字母，单词wine就会变成dine、fine、line、mine、nine、pine或者vine。替换第三个字母，wine就会变成wide、wife、wipe或者wire等。替换第四个字母，wine会变成wind、wing、wink或wins等。于是，仅仅通过替换wine中的一个字母，就得到了15个不同的单词。事实上，可以得到20个以上的单词。其中的一部分则是比较晦涩的单词。假设我们想要写一个程序，来找到所有的可以通过替换其中的一个字母而得到至少15个其他单词的单词。假设我们有一本字典，其中包含大约89 000个不同长度的单词。大多数的单词都有6至11个字母。其分布如下：8205个6个字母的单词、11 989个7个字母的单词、13 672个8个字母的单词、13 014个9个字母的单词、11 297个10个字母的单词以及8617个11个字母的单词。（事实上，那些最具有可变性的是有3、4或5个字母的单词，而较长的单词却消耗最多的时间。）

最直接的策略就是使用map，其中，键为单词，值为通过对键进行单个字符替换就可以得到的那些单词的集合。图4-68的例程显示了最终得到的map（我们还需要为这一部分编写代码）是如何打印所需的答案的。代码使用const\_iterator来遍历map，访问由词和词的向量构成的对的项。在第8和9行的常量引用用来替换复杂的表达式以及避免不必要的复制。

```

1 void printHighChangeables( const map<string,vector<string> > & adjWords,
2                             int minWords = 15 )
3 {
4     map<string,vector<string> >::const_iterator itr;
5
6     for( itr = adjWords.begin( ); itr != adjWords.end( ); ++itr )
7     {
8         const pair<string,vector<string> > & entry = *itr;
9         const vector<string> & words = entry.second;
10
11         if( words.size( ) >= minWords )
12         {
13             cout << entry.first << " (" << words.size( ) << "):";
14             for( int i = 0; i < words.size( ); i++ )
15                 cout << " " << words[ i ];
16             cout << endl;
17         }
18
19     }
20 }

```

图4-68 给定一个map，其键为单词，值为指向只有一个字符不同的单词组的vector。输出具有minwords或更多个通过一个字符替换就可以得到其他单词的那些单词



问题的关键在于如何从一个包含89 000个单词的数组来构造map。图4-69中的例程是一个直接的函数，该函数检测是否除了一个字符的替换之外，两个单词相同。我们可以使用这个例程作为构造map的最简单的算法。这是一个使用蛮力测试所有的单词对的测试。该算法见图4-70。

168

```

1 // Returns true if word1 and word2 are the same length
2 // and differ in only one character.
3 bool oneCharOff( const string & word1, const string & word2 )
4 {
5     if( word1.length( ) != word2.length( ) )
6         return false;
7
8     int diffs = 0;
9
10    for( int i = 0; i < word1.length( ); i++ )
11        if( word1[ i ] != word2[ i ] )
12            if( ++diffs > 1 )
13                return false;
14
15    return diffs == 1;
16 }

```

图4-69 检测两个单词是否仅有一个字母不同的例程

```

1 // Computes a map in which the keys are words and values are vectors of words
2 // that differ in only one character from the corresponding key.
3 // Uses a quadratic algorithm.
4 map<string,vector<string> > computeAdjacentWords( const vector<string> & words )
5 {
6     map<string,vector<string> > adjWords;
7
8     for( int i = 0; i < words.size( ); i++ )
9         for( int j = i + 1; j < words.size( ); j++ )
10            if( oneCharOff( words[ i ], words[ j ] ) )
11                {
12                    adjWords[ words[ i ] ].push_back( words[ j ] );
13                    adjWords[ words[ j ] ].push_back( words[ i ] );
14                }
15
16    return adjWords;
17 }

```

图4-70 用来计算map的函数。map的键为单词，值为指向只有一个字符不同的单词组的vector。这个版本的函数对89 000个单词的字典的运行时间是6.5 min

如果找到只有一个字符不同的单词对，就在第12和13行更新map。在第12行使用adj[str]给出只有一个字符与str不同的单词的vector。如果之前见过str，那么str在map中，我们仅需要调用push\_back添加新的单词到map中相应的vector中。如果以前没见过str，那么operator[]操作就将其放到map中，此时的vector大小为0，并返回这个vector，于是push\_back将vector的大小更新为1。总而言之，这是一个超级老套的惯用例程，用来维护其值为集合的map。

这个算法的问题在于其速度很慢，在我们的计算机上消耗6.5 min。一个显而易见的改进方案就是避免比较不同长度的单词。这可以通过将单词按不同长度进行分组来做到，然后再对每一个组使用前面的算法。

也可以使用第二个map来实现这个思想。这里的键是一个反映单词长度的整数，值是所有具有这个长度的单词的集合。可以使用vector来存储每个集合，然后应用相同的惯用例程。代码见图4-71。第8行是第二个map的声明，第11和12行添加map，然后使用一个额外的循环来迭代每



个单词组。与第一个算法相比较，第二个算法仅在编写代码时困难一些，但运行仅需要77 s，大概比第一个算法快6倍。

```

1 // Computes a map in which the keys are words and values are vectors of words
2 // that differ in only one character from the corresponding key.
3 // Uses a quadratic algorithm, but speeds things up a little by
4 // maintaining an additional map that groups words by their length.
5 map<string,vector<string>> computeAdjacentWords(const vector<string> & words)
6 {
7     map<string,vector<string>> adjWords;
8     map<int,vector<string>> wordsByLength;
9
10    // Group the words by their length
11    for( int i = 0; i < words.size(); i++ )
12        wordsByLength[ words[ i ].length() ].push_back( words[ i ] );
13
14    // Work on each group separately
15    map<int,vector<string>>::const_iterator itr;
16    for( itr = wordsByLength.begin(); itr != wordsByLength.end(); ++itr )
17    {
18        const vector<string> & groupsWords = itr->second;
19
20        for( int i = 0; i < groupsWords.size(); i++ )
21            for( int j = i + 1; j < groupsWords.size(); j++ )
22                if( oneCharOff( groupsWords[ i ], groupsWords[ j ] ) )
23                {
24                    adjWords[ groupsWords[ i ] ].push_back( groupsWords[ j ] );
25                    adjWords[ groupsWords[ j ] ].push_back( groupsWords[ i ] );
26                }
27    }
28
29    return adjWords;
30 }

```

图4-71 用来计算map的函数。map的键为单词，值为指向只有一个字符不同的单词组的vector。该函数将单词按长度分组。这个版本的函数对89 000个单词的字典运行时间是77s

第三个算法更复杂一些，使用了附加的map。与以前一样，将单词按长度分组，然后对每个组分别操作。为研究该算法的运行，假设现在运行在长度为4的单词组上。首先，我们想要找到形如wine和nine的只有一个字符不同的单词对。实现的一个方法如下：对每一个长度为4的单词，删除第一个字母，保留剩下三个字母的样本。生成一个map，其中的键是这个样本，其值是所有的有这个样本的单词的vector。例如，考虑四字母单词组的第一个字母，样本"ine"对应"dine"、"fine"、"wine"、"nine"、"mine"、"vine"、"pine"、"line"。样本"oot"对应"boot"、"foot"、"hoot"、"loot"、"soot"、"zoot"。最后这个map的值，即每一个单独的vector形成了一个单词组，在这个组中的每一个单词都可以通过一个字符的替换变成其他的单词，于是，最后一个map就构造出来。很容易遍历和添加项到所处理的原始的map中。然后我们使用一个新map来处理四字母单词组的第二个字母，然后是第三个字母，最后是第四个字母。

一般程序的构架如下：

```

for each group g, containing words of length len
    for each position p ( ranging from 0 to len-1)
    {
        Make an empty map<string,vector<string>> repsToWords
        for each word w
        {

```

```

        Obtain w's representative by removing position p
        Update repsToWords
    }
    Use cliques in repsToWords to update adjWords map
}

```

图4-72是这个算法的实现。运行时间提高到5 s。很有趣的是，虽然使用了附加的map使得算法更快，而且语法相对整齐，但代码并没有用到map的键始终保持排序状态这个事实。

```

1 // Computes a map in which the keys are words and values are vectors of words
2 // that differ in only one character from the corresponding key.
3 // Uses an efficient algorithm that is O(NlogN) with a map.
4 map<string,vector<string>> computeAdjacentWords( const vector<string> & words )
5 {
6     map<string,vector<string>> adjWords;
7     map<int,vector<string>> wordsByLength;
8
9     // Group the words by their length
10    for( int i = 0; i < words.size(); i++ )
11        wordsByLength[ words[ i ].length() ].push_back( words[ i ] );
12    // Work on each group separately
13    map<int,vector<string>>::const_iterator itr;
14    for( itr = wordsByLength.begin(); itr != wordsByLength.end(); ++itr )
15    {
16        const vector<string> & groupsWords = itr->second;
17        int groupNum = itr->first;
18
19        // Work on each position in each group
20        for( int i = 0; i < groupNum; i++ )
21        {
22            // Remove a character in given position, computing representative.
23            // Words with same representatives are adjacent; so populate a map
24            map<string,vector<string>> repToWord;
25
26            for( int j = 0; j < groupsWords.size(); j++ )
27            {
28                string rep = groupsWords[ j ];
29                rep.erase( i, 1 );
30                repToWord[ rep ].push_back( groupsWords[ j ] );
31            }
32
33            // and then look for map values with more than one string
34            map<string,vector<string>>::const_iterator itr2;
35            for( itr2 = repToWord.begin(); itr2 != repToWord.end(); ++itr2 )
36            {
37                const vector<string> & clique = itr2->second;
38                if( clique.size() >= 2 )
39                    for( int p = 0; p < clique.size(); p++ )
40                        for( int q = p + 1; q < clique.size(); q++ )
41                        {
42                            adjWords[ clique[ p ] ].push_back( clique[ q ] );
43                            adjWords[ clique[ q ] ].push_back( clique[ p ] );
44                        }
45            }
46        }
47    }
48    return adjWords;
49 }

```

图4-72 用来计算map的函数。map的键为单词，值为指向只有一个字符不同的单词组的vector。这个版本的函数对89 000个单词的字典运行时间是5 s

同样地，一个支持map操作但是不保证排序的数据结构可以运行得更好是很有可能，因为所要做的事更少。第5章探讨了这种可能性。一些C++库包括例如hash\_map的类，但这不是C++标准库中的部分。

172

## 小结

本章介绍了树在操作系统、编译器设计以及查找中的应用。表达式树是更一般的结构即所谓的分析树（parse tree）的一个小例子，分析树是编译器设计中的核心数据结构。分析树不是二叉树，而是表达式树相对简单的扩充（不过，建立分析树的算法却不是那么简单）。

查找树在算法设计中是非常重要的。它们几乎支持所有有用的操作，而其对数平均开销很小。查找树的非递归实现多少要快一些，但是递归实现更讲究、更精彩，而且更易于理解和调试。查找树的问题在于，其性能严重地依赖于随机的输入。如果情况不是这样（不是随机的），则运行时间会显著增加，查找树会成为昂贵的链表。

本章还介绍了处理这个问题的几个方法。AVL树要求所有结点的左子树与右子树的高度最多相差1，这就保证了树不至于太深。不改变树的操作（但插入操作改变树）都可以使用标准二叉查找树的程序。改变树的操作必须将树恢复。这多少有些复杂，特别是在删除时。我们叙述了在以 $O(\log N)$ 的时间插入后如何将树恢复。

我们还考察了伸展树。伸展树中的结点可以达到任意深度，但是在每次访问之后树又以多少有些神秘的方式被调整。实际效果是，任意连续 $M$ 次操作花费 $O(M \log N)$ 时间，它与平衡树花费的时间相同。

与2路树即二叉树不同，B树是平衡 $M$ 路树，它能很好地适应有磁盘操作的情况；一个特殊情形是2-3树（ $M=3$ ），它是实现平衡查找树的另一种方法。

在实践中，所有平衡树方案的运行时间对于插入和删除操作（除查找稍微快一些外）都不如简单二叉查找树省时（差一个常数因子），但这一般说来是可以接受的，它防止轻易得到最坏情形的输入。第12章将讨论另外一些查找树数据结构并给出详细的实现方法。

最后注意：通过将一些元素插入到查找树然后执行一次中序遍历，我们得到的是排过顺序的元素。这给出排序的一种 $O(M \log N)$ 算法，如果使用任何成熟的查找树则它就是最坏情形的界。我们将在第7章看到一些更好的方法，不过，这些方法的时间界都不可能更低。

## 练习

练习4.1到练习4.3参考图4-73中的树。

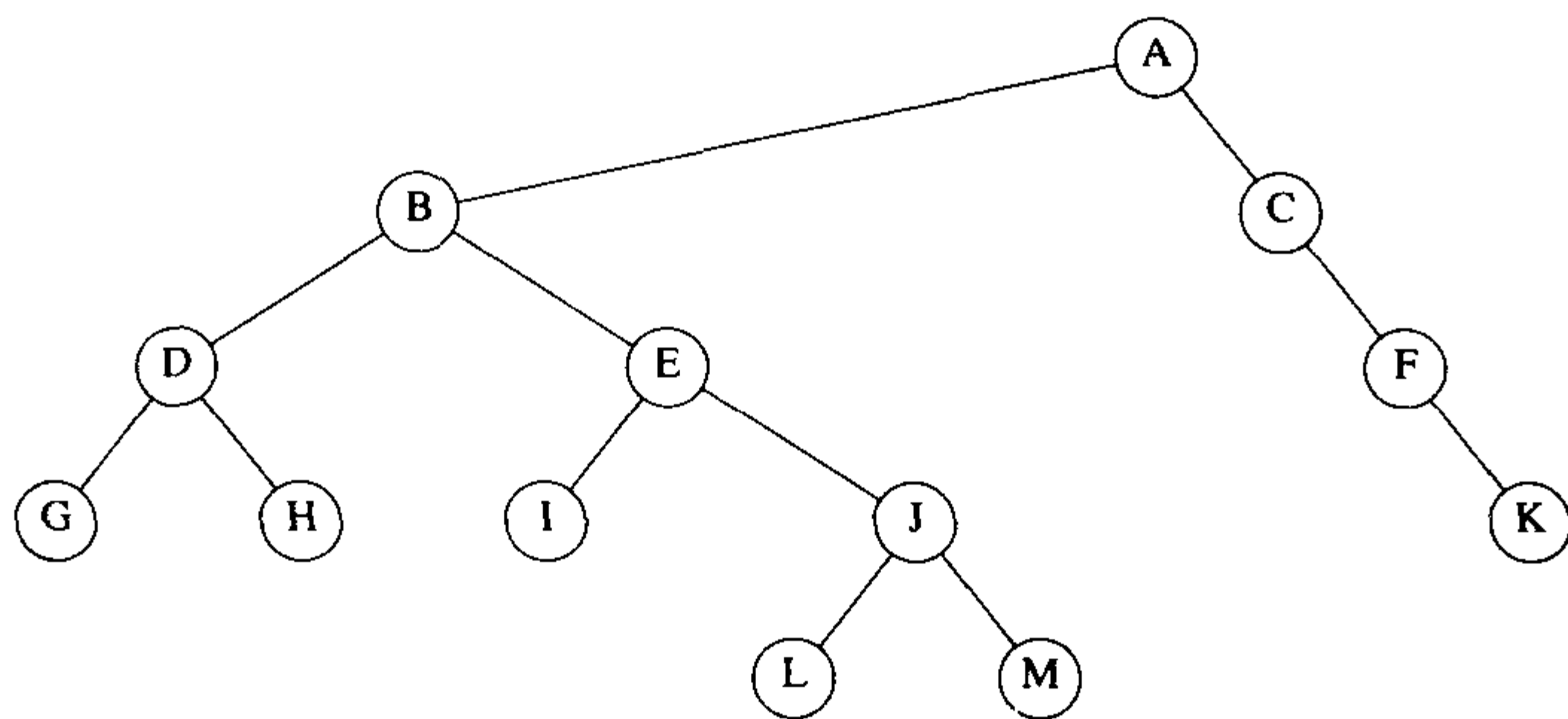


图4-73 练习4.1到练习4.3所用的树

- 4.1 对于图4-73中的树：
- 哪个结点是根？
  - 哪些结点是树叶？
- 4.2 对于图4-73中树上的每一个结点：
- 指出它的父结点。
  - 列出它的儿子。
  - 列出它的兄弟。
  - 计算它的深度。
  - 计算它的高度。
- 4.3 图4-73中树的深度是多少？
- 4.4 证明：在 $N$ 个结点的二叉树中，存在 $N+1$ 个NULL链代表 $N+1$ 个儿子。
- 4.5 证明：在高度为 $h$ 的二叉树中，结点的最大个数是 $2^{h+1}-1$ 。
- 4.6 满结点 (full node) 是具有两个儿子的结点。证明满结点的个数加1等于非空二叉树的树叶的个数。
- 4.7 设二叉树有树叶 $l_1, l_2, \dots, l_M$ ，各树叶的深度分别是 $d_1, d_2, \dots, d_M$ 。证明： $\sum_{i=1}^M 2^{-d_i} \leq 1$ 并确定何时等号成立。
- 4.8 给出对应图4-74中的树的前缀表达式、中缀表达式以及后缀表达式。

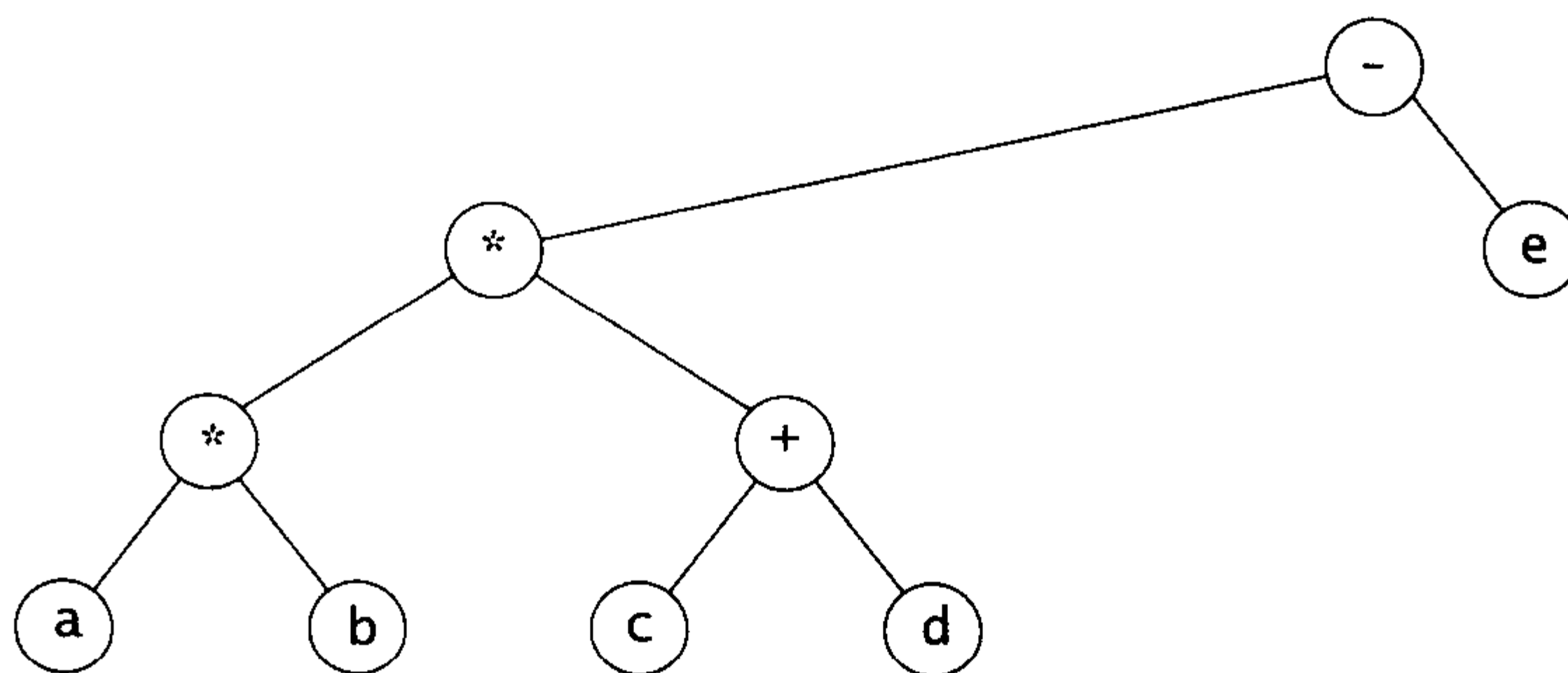


图4-74 练习4.8中的树

- 4.9
- 指出将3, 1, 4, 6, 9, 2, 5, 7插入到初始为空的二叉查找树中的结果。
  - 指出删除根后的结果。
- 4.10 令 $f(N)$ 是二叉查找树中的满结点个数的平均值。
- 确定 $f(0)$ 和 $f(1)$ 的值。
  - 证明：当 $N>1$ 时，下式成立：

$$f(N) = \frac{N-2}{N} + \frac{1}{N} \sum_{i=0}^{N-1} (f(i) + f(N-i-1))$$

- 证明：(用归纳法)  $f(N)=(N-2)/3$ 是b问中方程的解，其初始条件在a问中。
  - 用练习4.6中的结果来确定一颗二叉查找树中的树叶的平均个数。
- 4.11 编写一个set类的实现。使用相关的使用二叉查找树的迭代器。给每一个结点添加一个至父结点的链接。
- 4.12 编写一个map类的实现。其存储的数据成员类型为`set<Pair<KeyType, ValueType>>`。
- 4.13 编写一个set类的实现。使用相关的使用二叉查找树的迭代器。给每一个结点添加一个到下一个最小和最大结点的链接。为使代码更简单，添加一个头结点和一个尾结点。这两个结点不在二叉查找树中，但有助于使代码的链表部分更简单。
- 4.14 设要做一个实验来验证由随机insert/remove操作对可能引起的问题。这里有一个策略，它不是完全随机的，但却是足够封闭的。通过插入从1到 $M = \alpha N$ 之间随机选出的 $N$ 个元素来建立一棵



具有 $N$ 个元素的树。然后执行 $N^2$ 对先插入后删除的操作。假设存在例程`randomInteger(a, b)`，它返回一个在 $a$ 和 $b$ 之间（包括 $a$ 、 $b$ ）的均匀随机整数。

- 解释如何生成在1和 $M$ 之间的一个随机整数，该整数不在树上（从而随机插入可以进行）。用 $N$ 和 $\alpha$ 来表示这个操作的运行时间。
- 解释如何生成在1和 $M$ 之间的一个随机整数，该整数已经存在于树上（从而随机删除可以进行）。这个操作的运行时间是多少？
- $\alpha$ 的最佳选择值是什么？为什么？

4.15 编写一个程序，凭经验评估下列删除具有两个儿子的结点的各种方法：

- 用 $T_L$ 中最大结点 $X$ 来代替，递归地删除 $X$ 。
- 交替地用 $T_L$ 中最大的结点以及 $T_R$ 中最小的结点来代替，并递归地删除适当的结点。
- 随机地选用 $T_L$ 中最大的结点或 $T_R$ 中最小的结点来代替（递归地删除适当的结点）。

哪种方法给出最好的平衡？哪种在处理整个操作序列过程中花费最少的CPU时间？

4.16 重做二叉查找树类以实现懒惰删除。注意，这将影响所有的例程。特别具有挑战性的是`findMin`和`findMax`，它们现在必须递归地完成。

\*\*4.17 证明，随机二叉查找树的深度（最深的结点的深度）平均为 $O(\log N)$ 。

4.18 \* a. 给出高度为 $h$ 的AVL树的结点的最少个数的精确表达式。

b. 高度为15的AVL树中结点的最小个数是多少？

4.19 指出将2, 1, 4, 5, 9, 3, 6, 7插入到初始空AVL树后的结果。

\*4.20 依次将键1, 2, ...,  $2^k - 1$ 插入到一棵初始空AVL树中。证明所得到的树是完全平衡的。

4.21 写出实现AVL单旋转和双旋转的其余的过程。

4.22 设计一个线性时间算法，该算法检验AVL树中的高度信息是否被正确保留并且平衡性质成立。

4.23 写出向AVL树进行插入的非递归函数。

\*4.24 如何在AVL树中实现（非懒惰）删除？

4.25 a. 为了存储一棵 $N$ -结点的AVL树中一个结点的高度，每个结点需要多少位（bit）？

b. 使8位高度计数器溢出的最小AVL树是什么？

4.26 写出执行双旋转的函数，其效率要超过做两个单旋转。

4.27 指出依序访问图4-75的伸展树中的键3, 9, 1, 5后的结果。

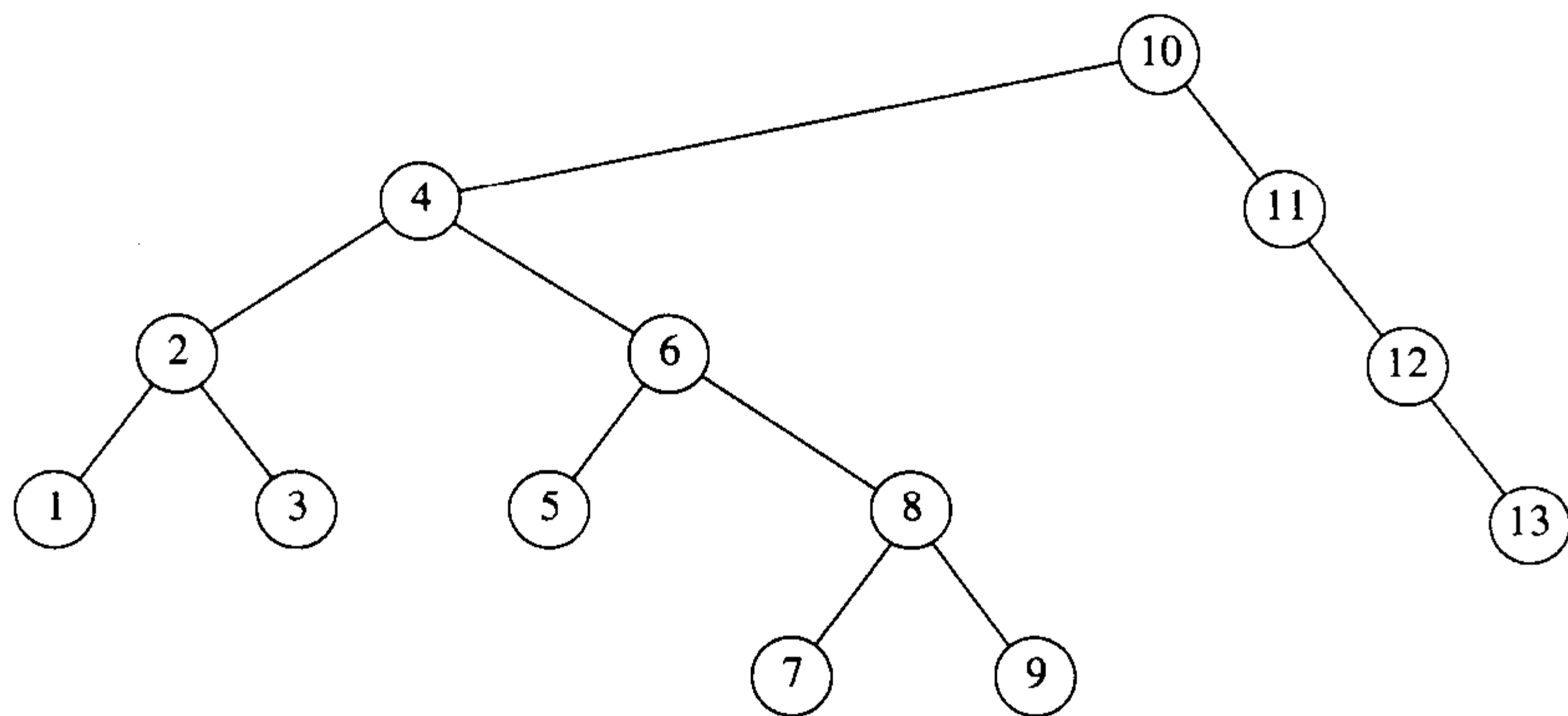


图4-75 练习4.27中的树

4.28 指出在前一道练习所得到的伸展树中删除具有键为6的元素后的结果。

4.29 a. 证明如果按顺序访问伸展树中的所有结点，则所得到的树由一连串的左儿子组成。

\*\*b. 证明如果按顺序访问伸展树中的所有结点，则总的访问时间是 $O(N)$ ，与初始树无关。

4.30 编写一个程序对伸展树执行随机操作。计算所执行的总的旋转次数。与AVL树和非平衡二叉查找

树相比，其运行时间如何？

4.31 编写一些高效率的函数，只使用指向二叉树 $T$ 的根的指针，并计算：

- a.  $T$ 中结点的个数。
- b.  $T$ 中树叶的片数。
- c.  $T$ 中满结点的个数。

所得例程的运行时间是多少？

4.32 设计一个递归的线性算法，该算法测试一棵二叉树是否在每一个结点都满足查找树的序的性质。

4.33 编写一个递归函数，该函数使用指向树 $T$ 的根结点的指针而返回指向从 $T$ 中删除所有树叶所得到的树的根结点的指针。

4.34 写出生成一棵 $N$ 结点随机二叉查找树的函数，该树具有从1直到 $N$ 的不同的键。你所编写的例程的运行时间是多少？

4.35 写出生成具有最少结点的高度为 $h$ 的AVL树的函数，该函数的运行时间是多少？

4.36 编写一个函数，使它生成一棵具有键从1直到 $2^{h+1}-1$ 且高为 $h$ 的理想平衡二叉查找树（perfectly balanced binary search tree）。该函数的运行时间是多少？

4.37 编写一个函数以二叉查找树 $T$ 和两个有序的键 $k_1$ 和 $k_2$ 作为输入，其中 $k_1 \leq k_2$ ，并打印树中所有满足 $k_1 \leq \text{Key}(X) \leq k_2$ 的元素 $X$ 。除去可以被排序外，不对键的类型做任何假设。所写的程序应该以平均时间 $O(K + \log N)$ 运行，其中 $K$ 是所打印的键的个数。确定你的算法的运行时间界。

4.38 本章中一些更大的二叉树是由一个程序自动生成的。这可以通过给树的每一个结点指定坐标 $(x, y)$ ，围绕每个坐标点画一个圆圈（在某些图片中这可能很难看清），并将每个结点连到它的父结点上进行。假设在内存中存有一棵二叉查找树（或许是由上面的一个例程生成的），并设每个结点都有两个附加的字段来存放坐标。

- a. 坐标 $x$ 可以通过指定中序遍历数来计算。对树中的每个结点写出一个这样的例程。
- b. 坐标 $y$ 可以通过结点深度的负值算出。对树中的每个结点写出一个这样的例程。
- c. 若使用某个虚拟的单位表示，则所画图形的具体尺寸是多少？如何调整单位使得所画的树总是高大约为宽的2/3？
- d. 证明，使用这个系统没有交叉线出现，同时，对于任意结点 $X$ ， $X$ 的左子树的所有元素都出现在 $X$ 的左边， $X$ 的右子树的所有元素都出现在 $X$ 的右边。

4.39 编写一个通用的画树程序，该程序将把一棵树转变成下列的图-汇编语言指令：

- a. *Circle*( $X, Y$ )
- b. *DrawLine*( $i, j$ )

第一个指令在 $(X, Y)$ 处画一个圆，而第二个指令则连接第 $i$ 个圆和第 $j$ 个圆（圆以所画的顺序编号）。你或者把它写成一个程序并确定某种输入语言，或者把它写成一个函数，该函数可以被任何程序调用。你的程序的运行时间是多少？

4.40 编写一个例程以层序列出二叉树的结点。先列出根，然后列出深度为1的那些结点，再列出深度为2的结点，依此类推。必须要在线性时间内完成这个工作。证明该时间界。

4.41 \* a. 写出向一棵B树进行插入的例程。

\* b. 写出从一棵B树执行删除的例程。当一个项被删除时，是否有必要更新内部结点的信息？

\* c. 修改你的插入例程使得如果想要向一个已经有 $M$ 项的结点添加元素，则在分裂该结点以前要执行搜索具有少于 $M$ 个儿子的兄弟的工作。

4.42  $M$ 阶B\*树（B\*-tree）是其每个内部结点的儿子数在 $2M/3$ 和 $M$ 之间的B树。描述一种向B\*树执行插入的方法。

4.43 指出如何用儿子/兄弟链实现方法表示图4-76中的树。

4.44 编写一个过程来遍历一棵用儿子/兄弟链存储的树。

4.45 如果两棵二叉树或者都是空树，或者非空且具有相似的左子树和右子树，则这两棵二叉树是相似

177

178

179

的。编写一个函数以确定是否两棵二叉树是相似的。你的程序的运行时间如何？

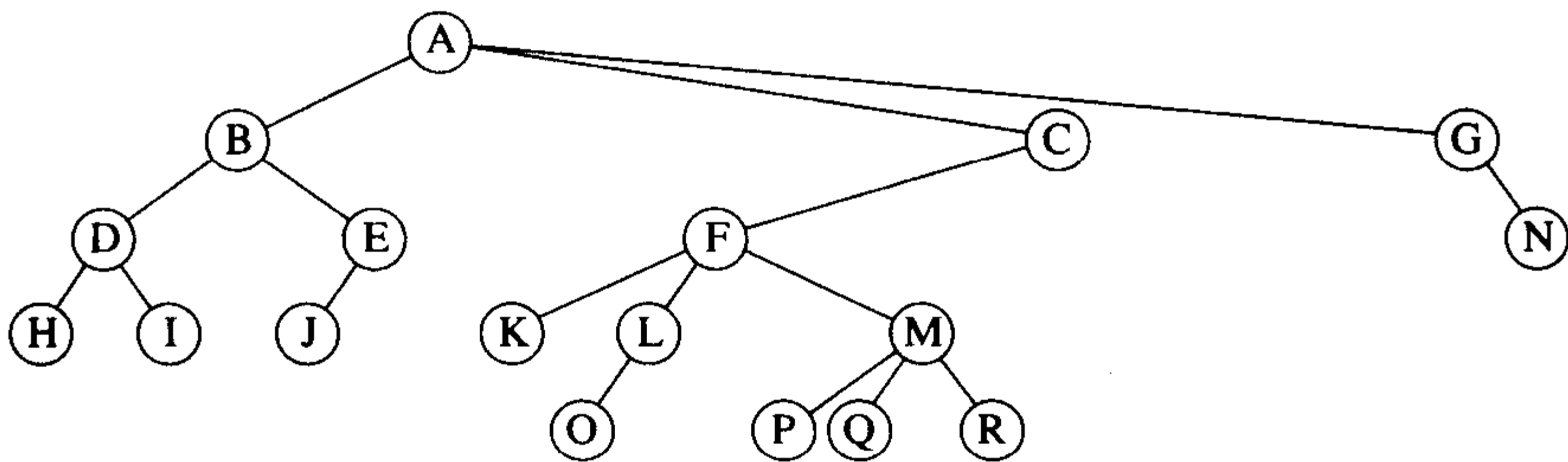


图4-76 练习4.43中的树

- 4.46 如果树 $T_1$ 通过交换其（某些）结点的左右儿子变换成树 $T_2$ ，则称树 $T_1$ 和 $T_2$ 是同构的（isomorphic）。例如，图4-77中的两棵树是同构的，因为交换A、B、G的儿子而不交换其他结点的儿子后这两棵树是相同的。

- a. 给出一个多项式时间算法以决定是否两棵树是同构的。
- \* b. 你的程序的运行时间是多少（存在一个线性的解决方案）？



图4-77 两棵同构的树

- 4.47 \*a. 证明，经过一些AVL单旋转，任意二叉查找树 $T_1$ 可以变换成另一棵（具有相同项的）查找树 $T_2$ 。  
\*b. 给出一个算法，平均用 $O(N \log N)$ 次旋转完成这种变换。  
\*\*c. 证明该变换在最坏的情形下可以用 $O(N)$ 次旋转完成。
- 4.48 设我们想要把运算findKth添加到指令集中去。该运算findKth(k)返回树的第k个最小项。假设所有的项都是互异的。解释如何修改二叉查找树以平均 $O(\log N)$ 时间支持这种运算，而又不影响任何其他操作的时间界。
- 4.49 由于具有N个结点的二叉查找树有 $N + 1$ 个NULL指针，因此在二叉查找树中指定给链接信息的空间的一半被浪费了。设若一个结点有一个NULL左儿子，我们使它的左儿子链接到其中序前驱元（inorder predecessor），若一个结点有一个NULL右儿子，我们让它的右儿子链接到其中序后继元（inorder successor）。这就叫作线索树（threaded tree），而附加的链就叫作线索（thread）。  
a. 我们如何能够从实际儿子的指针中区分出线索？  
b. 编写执行向上面描述的方式形成的线索树进行插入的例程和删除的例程。  
c. 使用线索树的优点是什么？
- 4.50 编写一个程序，该程序读C++源代码文件并以字母顺序输出所有的标识符（即变量名而非关键字，并且这些变量名不是从注释和串常数中找出的）。每个标识符要和它所在的行的行号一起输出。
- 4.51 为一本书生成一个索引。输入文件由一组索引项组成。每行由串ix: 组成，其后跟着一个索引项的名字，封在大括号内，后面是封在大括号内的页号。索引项名字中的每个!代表一个子层（sub-level）。符号|(代表一个范围的开始，而)|则代表这个范围的结束。偶尔这个范围是在同一页中。在这种情形下只输出一个页号。在其他情况下不要套叠，否则你自己就扩大了范围。例如，图4-78显示了样本输入，而图4-79则显示了对应的输出。

|                            |     |
|----------------------------|-----|
| IX: {Series {}             | {2} |
| IX: {Series!geometric {}   | {4} |
| IX: {Euler's constant}     | {4} |
| IX: {Series!geometric })}  | {4} |
| IX: {Series!arithmetic {}  | {4} |
| IX: {Series!arithmetic })} | {5} |
| IX: {Series!harmonic {}    | {5} |
| IX: {Euler's constant}     | {5} |
| IX: {Series!harmonic })}   | {5} |
| IX: {Series })}            | {5} |

图4-78 练习4.51的样本输入

|                        |
|------------------------|
| Euler's constant: 4, 5 |
| Series: 2-5            |
| arithmetic: 4-5        |
| geometric: 4           |
| harmonic: 5            |

图4-79 练习4.51的样本输出

## 参考文献

关于二叉查找树的更多信息，特别是树的数学性质可以在Knuth[22]和[23]的两本书中找到。

有几篇论文讨论由二叉查找树中的有偏删除（biased deletion）算法引起的平衡不足问题。Hibbard的论文[19]提出原始删除算法并确立了一次删除保持树的随机性。文献[20]和[5]分别对只有三个结点的树和四个结点的树进行了全面的分析。Eppinger的论文[14]提供了非随机性的早期经验性的证据，而Culberson和Munro的论文[10]和[11]则提供了某些解析论证（但不是对混杂插入和删除的一般情形的完整证明）。

AVL树由Adelson-Velskii和Landis[1]提出。AVL树的模拟结果以及高度的不平衡允许最多到 $k$ 的各种变体，在[21]中讨论。AVL树的删除算法可以在[23]中找到。在AVL树中平均搜索开销的分析是不完全的，但是，[24]中包含了某些结果。

文献[3]和文献[8]考虑了类似本书4.5.1节类型的自调整树。伸展树在[28]中进行了描述。

B树首先出现在[6]中。原始论文中所描述的实现方法允许数据存储在内部结点和树叶上。我们描述过的数据结构有时叫作B+树。[9]对不同类型的B树进行了综合分析。[17]报告了各种方案的经验性结果。2-3树和B树的分析可以在[4]、[13]以及[32]中找到。

练习4.17使人误以为很难。一种解法可以在[15]中找到。练习4.29取自[32]。在练习4.42中描述的B\*树的信息可以在[12]中找到。练习4.46取自文献[2]。练习4.47的解法使用 $2N - 6$ 次旋转，该解法在[29]中给出。按照练习4.49的方式使用的线索首先在[27]中提出。 $k$ -d树最早是在[7]中提出来的，将在本书第12章进行讨论，它处理多维数据。

另外一些流行的平衡查找树是红黑树[18]和加权平衡树[26]。在第12章可以找到更多的平衡树方案，此外也可以在著作[16]、[25]以及[30]中找到。

181

1. G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet. Mat. Doklady*, 3 (1962), 1259-1263.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
3. B. Allen and J. I. Munro, "Self Organizing Search Trees," *Journal of the ACM*, 25 (1978), 526-535.
4. R. A. Baeza-Yates, "Expected Behaviour of B<sup>+</sup>-trees under Random Insertions," *Acta Informatica*, 26 (1989), 439-471.
5. R. A. Baeza-Yates, "A Trivial Algorithm Whose Analysis Isn't: A Continuation," *BIT*, 29 (1989), 88-113.
6. R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," *Acta*



- Informatica*, 1 (1972), 173-189.
7. J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, 18 (1975), 509-517.
  8. J. R. Bitner, "Heuristics that Dynamically Organize Data Structures," *SIAM Journal on Computing*, 8 (1979), 82-110.
  9. D. Comer, "The Ubiquitous B-tree," *Computing Surveys*, 11 (1979), 121-137.
  10. J. Culberson and J. I. Munro, "Explaining the Behavior of Binary Search Trees under Prolonged Updates: A Model and Simulations," *Computer Journal*, 32 (1989), 68-75.
  11. J. Culberson and J. I. Munro, "Analysis of the Standard Deletion Algorithms in Exact Fit Domain Binary Search Trees," *Algorithmica*, 5 (1990) 295-311.
  12. K. Culik, T. Ottman, and D. Wood, "Dense Multiway Trees," *ACM Transactions on Database Systems*, 6 (1981), 486-512.
  13. B. Eisenbath, N. Ziviana, G. H. Gonnet, K. Melhorn, and D. Wood, "The Theory of Fringe Analysis and Its Application to 2-3 Trees and B-trees," *Information and Control*, 55 (1982), 125-174.
  14. J. L. Eppinger, "An Empirical Study of Insertion and Deletion in Binary Search Trees," *Communications of the ACM*, 26 (1983), 663-669.
  15. P. Flajolet and A. Odlyzko, "The Average Height of Binary Trees and Other Simple Trees," *Journal of Computer and System Sciences*, 25 (1982), 171-213.
  16. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2nd ed., Addison-Wesley, Reading, Mass., 1991.
  17. E. Gudes and S. Tsur, "Experiments with B-tree Reorganization," *Proceedings of ACM SIGMOD Symposium on Management of Data* (1980), 200-206.
  18. L.J. Guibas and R. Sedgwick, "A Dichromatic Framework for Balanced Trees," *Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science* (1978), 8-21.
  19. T. H. Hibbard, "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting," *Journal of the ACM*, 9 (1962), 13-28.
  20. A. T. Jonassen and D. E. Knuth, "A Trivial Algorithm Whose Analysis Isn't," *Journal of Computer and System Sciences*, 16 (1978), 301-322.
  21. P. L. Karlton, S. H. Fuller, R. E. Scroggs, and E. B. Kaehler, "Performance of Height Balanced Trees," *Communications of the ACM*, 19 (1976), 23-28.
  22. D. E. Knuth, *The Art of Computer Programming: Vol. 1: Fundamental Algorithms*, 3rd ed., Addison-Wesley, Reading, Mass., 1997.
  23. D.E. Knuth, *The Art of Computer Programming: Vol. 3: Sorting and Searching*, 2nd ed., Addison-Wesley, Reading, Mass., 1998.
  24. K. Melhorn, "A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions," *SIAM Journal of Computing*, 11 (1982), 748-760.
  25. K. Melhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.
  26. J. Nievergelt and E. M. Reingold, "Binary Search Trees of Bounded Balance," *SIAM Journal on Computing*, 2 (1973), 33-43.
  27. A.J. Perlis and C. Thornton, "Symbol Manipulation in Threaded Lists," *Communications of the ACM*, 3 (1960), 195-204.
  28. D. D. Sleator and R. E. Tarjan, "Self-adjusting Binary Search Trees," *Journal of the ACM*, 32 (1985), 652-686.
  29. D. D. Sleator, R. E. Tarjan, and W. P. Thurston, "Rotation Distance, Triangulations, and Hyperbolic Geometry," *Journal of the AMS* (1988), 647-682.
  30. H. F. Smith, *Data Structures—Form and Function*, Harcourt Brace Jovanovich, Orlando, Fla., 1987.
  31. R. E. Tarjan, "Sequential Access in Splay Trees Takes Linear Time," *Combinatorica*, 5 (1985), 367-378.
  32. A. C. Yao, "On Random 2-3 Trees," *Acta Informatica*, 9 (1978), 159-170.

散 列

第4章讨论了查找树ADT，它允许对一组元素进行各种操作。本章讨论散列表（hash table）ADT，不过它只支持二叉查找树所允许的一部分操作。

散列表的实现常称为散列（hashing）。散列是一种用于以常数平均时间执行插入、删除和查找的技术。但是，那些需要元素间任何排序信息的树操作将不会得到有效的支持。因此，诸如 findMin、findMax以及在线性时间内按顺序打印整个表的操作都是散列所不支持的。

本章的中心数据结构是散列表。我们将：

- 看到实现散列表的几种方法。
- 分析比较这些方法。
- 介绍散列的多种应用。
- 将散列表和二叉查找树进行比较。

5.1 基本思想

理想的散列表数据结构只不过是一个包含一些项的具有固定大小的数组。第4章讨论过，查找一般是对项的某个部分（即数据成员）进行，这部分称为键（key）。例如，项可以由字符串（它可以作为键）和附加的数据成员组成（例如，姓名是大型雇员结构的一部分）。我们把表的大小记作 *TableSize*，并将其理解为散列数据结构的一部分而不仅仅是浮动于全局的某个变量。通常的习惯是让表从0到 *TableSize* - 1变化；稍后我们就会明白为什么要这样。

将每个键映射到从0到 *TableSize* - 1这个范围中的某个数，并且将其放到适当的单元中。这个映射就称为散列函数（hash function），理想情况下它应该运算简单并且应该保证任何两个不同的键映射到不同的单元。不过，这是不可能的，因为单元的数目是有限的，而键实际上是用不完的。因此，我们寻找一个散列函数，该函数要在单元之间均匀地分配键。图5-1是一个典型的理想情况。在这个例子中，john散列到3，phil散列到4，dave散列到6，mary散列到7。

|   |            |
|---|------------|
| 0 |            |
| 1 |            |
| 2 |            |
| 3 | john 25000 |
| 4 | phil 31250 |
| 5 |            |
| 6 | dave 27500 |
| 7 | mary 28200 |
| 8 |            |
| 9 |            |

这就是散列的基本思想。剩下的问题则是要选择一个函数，决定当两个键散列到同一个值的时候（称为冲突（collision））应该做什么以及如何确定散列表的大小。

5.2 散列函数

如果输入的键是整数，则一般合理的方法就是直接返回 “*Key mod Tablesize*”，除非*Key*碰巧



具有某些不理想的性质。在这种情况下，散列函数的选择需要仔细考虑。例如，若表的大小是10而键的个位都是0，则此时上述标准的散列函数就是一个不好的选择。其原因我们将在后面介绍，而为了避免上面那样的情况，好的办法通常是保证表的大小是素数。当输入的键是随机整数时，散列函数不仅运算简单而且键的分配也很均匀。

通常，键是字符串；在这种情形下，散列函数需要仔细选择。

一种选择方法是把字符串中字符的ASCII码值加起来。图5-2中的例程实现了这种策略。

```

1 int hash( const string & key, int tableSize )
2 {
3     int hashVal = 0;
4
5     for( int i = 0; i < key.length( ); i++ )
6         hashVal += key[ i ];
7
8     return hashVal % tableSize;
9 }

```

图5-2 一个简单的散列函数

图5-2中描述的散列函数实现起来简单而且能够很快地算出答案。不过，如果表很大，则函数就不会很好地分配键。例如，设 $TableSize = 10\,007$ （10 007是素数），并设所有的键至多8个字符长。由于ASCII字符的值最多是127，因此散列函数只能在0~1016之间取值，其中1016为 $127 \times 8$ 。显然这不是一种均匀的分配。

另一个散列函数如图5-3所示。这个散列函数假设 $Key$ 至少有3个字符。值27表示英文字母表的字母个数外加一个空格，而729是 $27^2$ 。该函数只考察前三个字符，但是，假如它们是随机的，而表的大小像前面那样还是10 007，那么我们会得到一个合理的均衡分布。可是，英文不是随机的。虽然3个字符（忽略空格）有 $26^3 = 17\,576$ 种可能的组合，但查验词汇量足够大的联机词典却揭示出：3个字母的不同组合数实际上只有2851。即使这些组合没有冲突，也不过只有表的28%被真正散列到。因此，虽然很容易计算，但是当散列表足够大的时候这个函数还是不适用的。

```

1 int hash( const string & key, int tableSize )
2 {
3     return ( key[ 0 ] + 27 * key[ 1 ] + 729 * key[ 2 ] ) % tableSize;
4 }

```

图5-3 另一个可能的散列函数——不是太好

图5-4列出了散列函数的第3种尝试。这个散列函数涉及键中的所有字符，并且一般可以分布得很好（它计算 $\sum_{i=0}^{KeySize-1} Key[KeySize-i-1] \cdot 37^i$ ，并将结果限制在适当的范围）。程序根据Horner法则计算一个（37的）多项式函数。例如，计算 $h_k = k_0 + 37k_1 + 37^2k_2$ 的另一种方式是借助于公式 $h_k = ((k_2) \times 37 + k_1) \times 37 + k_0$ 进行。Horner法则将其扩展到用于 $n$ 次多项式。

这个散列函数利用了允许溢出这个事实。这可能会引进负数，因此在末尾有附加的测试。

图5-4所描述的散列函数就表的分布而言未必是最好的，但是确实具有极其简单的优点而且速度也很快。如果键特别长，那么该散列函数计算起来将会花费过多的时间。在这种情况下通常的做法是不使用所有的字符。此时键的长度和性质将影响选择。例如，键可能是完整的街道地址，散列函数可以包括街道地址的几个字符，或许也包括城市名和邮政编码的几个字符。有些程序设计人员通过只使用奇数位置上的字符来实现他们的散列函数，这里有这么一层想法：用计算散列

函数节省下的时间来补偿由此产生的对均匀分布的函数的轻微干扰。

187

```

1  /**
2   * A hash routine for string objects.
3   */
4  int hash( const string & key, int tableSize )
5  {
6      int hashVal = 0;
7
8      for( int i = 0; i < key.length( ); i++ )
9          hashVal = 37 * hashVal + key[ i ];
10
11     hashVal %= tableSize;
12     if( hashVal < 0 )
13         hashVal += tableSize;
14
15     return hashVal;
16 }

```

图5-4 一个好的散列函数

剩下的主要编程细节是解决冲突。如果当一个元素在插入时与一个已经插入的元素散列到相同的值，那么就产生一个冲突，这个冲突需要消除。解决这种冲突的方法有几种，我们将讨论其中最简单的两种：分离链接法和开放定址法。

## 5.3 分离链接法

解决冲突的第一种方法通常称为分离链接法（separate chaining），其做法是将散列到同一个值的所有元素保留到一个链表中。可以使用标准库中表的实现方法。如果空间很紧，则更可取的方法是避免使用它们（因为这些表是双向链表，浪费空间）。本节假设键是前10个完全平方数并设散列函数就是 $hash(x) = x \bmod 10$ （表的大小不是素数，用在这里只是为了简单）。图5-5对此进行了更清晰的解释。

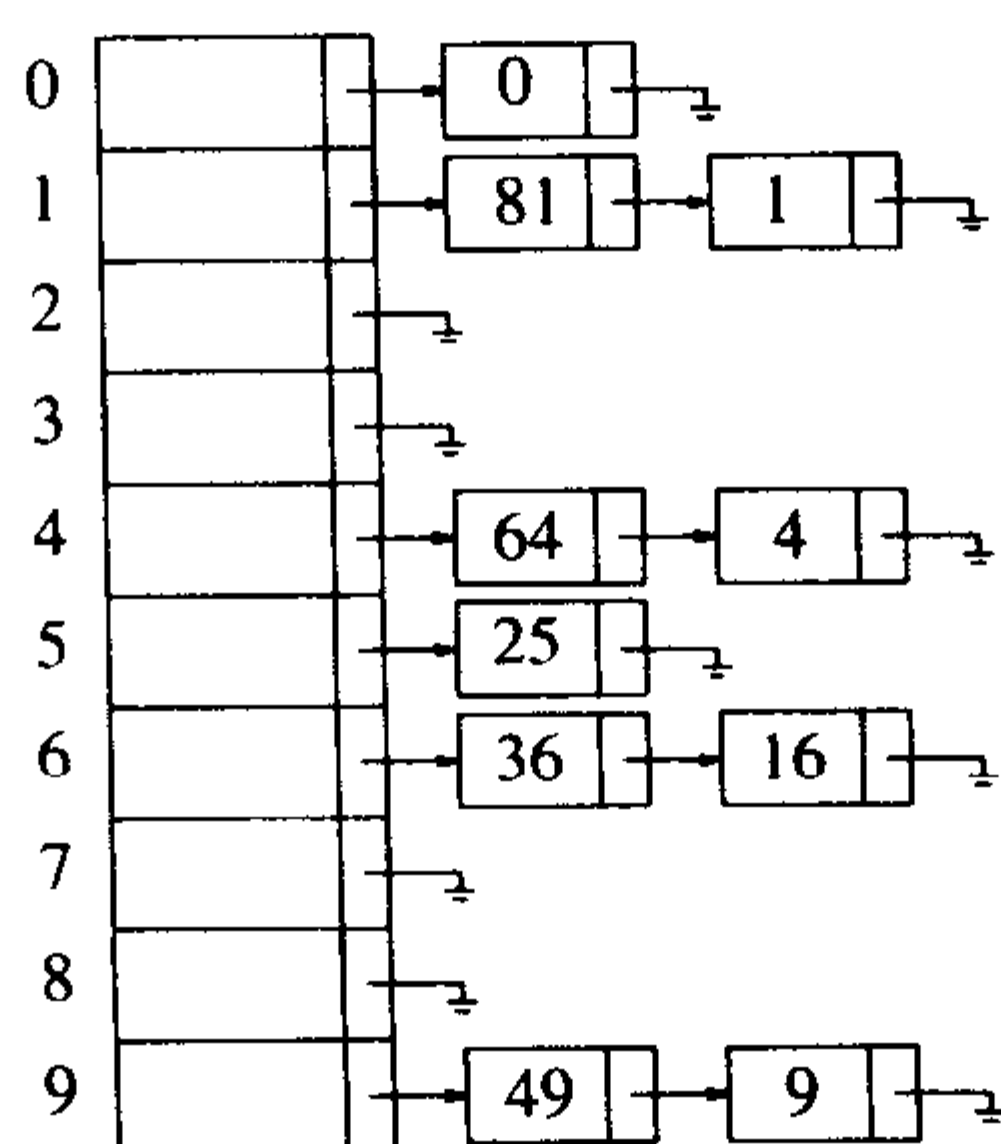


图5-5 分离链接散列表

为执行search，我们使用散列函数来确定究竟该遍历哪个链表。然后查找相应的表。为执行insert，我们检查相应的表来确定是否该元素已经在表中了（如果要插入重复元，那么通常要留出一个额外的数据成员，当出现匹配事件时这个数据成员增1）。如果这个元素是个新的元素，那么它将被插入到表的前端，这不仅因为方便，而且还因为最后插入的元素最有可能不久再次被



访问。

实现分离链接法所需要的类架构在图5-6中给出。散列表结构存储一个链表数组，这个数组在构造函数中指定。

```

1  template <typename HashedObj>
2  class HashTable
3  {
4  public:
5      explicit HashTable( int size = 101 );
6
7      bool contains( const HashedObj & x ) const;
8
9      void makeEmpty( );
10     void insert( const HashedObj & x );
11     void remove( const HashedObj & x );
12
13 private:
14     vector<list<HashedObj> > theLists;    // The array of Lists
15     int  currentSize;
16
17     void rehash( );
18     int myhash( const HashedObj & x ) const;
19 };
20
21 int hash( const string & key );
22 int hash( int key );

```

图5-6 分离链接散列表的类构架

类接口给出了一个语法点：在theLists的声明中，在两个>之间需要有一个空格。因为>>是C++的一个运算符，而且比>长，因此，>>会被认为是运算符。

就像二叉查找树只适合Comparable的对象一样，本章中的散列表只适合提供散列函数和相等性操作符（operator==或operator!=或两者都提供）的对象。这将自动包括int和string，因为在HashTable类中这些散列函数是作为全局的非类函数出现的。

这里不使用那些将对象和表的大小作为参数的散列函数，而是使用那些仅以对象为参数的散列函数，并且返回int。散列表类有一个私有的成员函数myhash，这个函数将结果分配到一个合适的数组索引中。图5-7所示为myHash的代码。

```

1  int myhash( const HashedObj & x ) const
2  {
3      int hashVal = hash( x );
4
5      hashVal %= theLists.size( );
6      if( hashVal < 0 )
7          hashVal += theLists.size( );
8
9      return hashVal;
10 }

```

图5-7 散列表的myHash成员函数

图5-8列出了一个可以存储在一般散列表中的Employee类，该类使用name成员作为键。Employee类通过提供相等性操作符和一个散列函数来实现HashedObj的需求。

```

1 // Example of an Employee class
2 class Employee
3 {
4     public:
5         const string & getName( ) const
6         { return name; }
7
8         bool operator==( const Employee & rhs ) const
9         { return getName( ) == rhs.getName( ); }
10        bool operator!=( const Employee & rhs ) const
11        { return !( *this == rhs; }
12
13        // Additional public members not shown
14
15    private:
16        string name;
17        double salary;
18        int    seniority;
19
20        // Additional private members not shown
21 };
22
23 int hash( const Employee & item )
24 {
25     return hash( item.getName( ) );
26 }

```

图5-8 可以作为HashedObj使用的类的一个例子

实现makeEmpty、contains和remove的程序代码见图5-9。

```

1 void makeEmpty( )
2 {
3     for( int i = 0; i < theLists.size( ); i++ )
4         theLists[ i ].clear( );
5 }
6
7 bool contains( const HashedObj & x ) const
8 {
9     const list<HashedObj> & whichList = theLists[ myhash( x ) ];
10    return find( whichList.begin( ), whichList.end( ), x ) != whichList.end( );
11 }
12
13 bool remove( const HashedObj & x )
14 {
15     list<HashedObj> & whichList = theLists[ myhash( x ) ];
16     list<HashedObj>::iterator itr = find( whichList.begin( ), whichList.end( ), x );
17
18     if( itr == whichList.end( ) )
19         return false;
20
21     whichList.erase( itr );
22     --currentSize;
23     return true;
24 }

```

图5-9 分离链接散列表的makeEmpty、contains和remove例程

下一个操作是插入例程。如果要插入的项已经存在，那么什么都不做；否则将其放至表的前端（见图5-10）。该元素可以放在表的任何地方；此处使用push\_back是最方便的。whichList

189  
191

是一个引用变量（关于引用变量的使用的讨论见1.5.4节的相关部分）。

```

1 bool insert( const HashedObj & x )
2 {
3     list<HashedObj> & whichList = theLists[ myhash( x ) ];
4     if( find( whichList.begin( ), whichList.end( ), x ) != whichList.end( ) )
5         return false;
6     whichList.push_back( x );
7
8     // Rehash; see Section 5.5
9     if( ++currentSize > theLists.size( ) )
10        rehash( );
11
12    return true;
13 }

```

图5-10 分离链接散列表的insert例程

除链表外，任何的方案也都有可能用来解决冲突现象；一棵二叉查找树，甚至另外一个散列表均可胜任，但我们期望的是如果散列表大而且散列函数好，那么所有的链表都应该短，因此，不值得去进行任何复杂的尝试。

我们定义散列表的装填因子（load factor） $\lambda$ 为散列表中的元素个数与散列表大小的比值。在上面的例子中， $\lambda=1.0$ 。表的平均长度为 $\lambda$ 。执行一次查找所需要的工作是计算散列函数值所需要的常数时间加上遍历表所用的时间。在一次不成功的查找中，要考察的结点数平均为 $\lambda$ 。成功的查找则需要遍历大约 $1 + (\lambda/2)$ 个链。为了看清这一点，注意，被搜索的表包含一个存储匹配的结点再加上0个或多个其他的结点。在 $N$ 个元素的表以及 $M$ 个链表中“其他结点”的期望个数为 $(N-1)/M = \lambda - 1/M$ ，它基本上就是 $\lambda$ ，因为认为 $M$ 很大。平均来看，一半的“其他结点”被搜索到，再结合匹配结点，我们得到 $1 + \lambda/2$ 个结点的平均查找开销。这个分析指出，散列表的大小实际上并不重要，而装填因子才是重要的。分离链接散列法的一般法则是使得表的大小尽量与预料的元素个数差不多（换句话说，让 $\lambda \approx 1$ ）。在图5-10的代码中，如果装填因子超过1，我们就使用第10行的rehash来扩充表的大小。rehash将在5.5节讨论。正如前面提到的，使表的大小是素数以保证一个好的分布，这也是一个好的想法。

## 5.4 不使用链表的散列表

分离链接散列算法的缺点是使用一些链表。由于给新单元分配地址需要时间（特别是在其他语言中），因此这就导致算法的速度有些减慢，同时算法实际上还要求第二种数据结构的实现。使用链表的另一个解决冲突的方法是当冲突发生时就尝试选择另外一个单元，直到找到空的单元。更正式地，单元 $h_0(x), h_1(x), h_2(x), \dots$ 依次进行试选，其中 $h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{TableSize}$ ，且 $f(0) = 0$ 。函数 $f$ 是冲突解决函数。因为所有的数据都要置入表内，所以使用这个方案所需要的表要比分离链接散列需要的表大。一般说来，对不使用分离链接法的散列表来说，其装填因子应该低于 $\lambda = 0.5$ 。我们称这样的表为探测散列表（probing hash tables）。下面考察三种常见的解决冲突的方法。

### 5.4.1 线性探测

在线性探测中，函数 $f$ 是 $i$ 的线性函数，一般情况下 $f(i) = i$ 。这相当于逐个探测每个单元（使用回绕）来查找出空单元。图5-11显示了使用与前面相同的散列函数将诸键{89, 18, 49, 58, 69}插

入到一个散列表中的情况，而此时的冲突解决方法就是 $f(i) = i$ 。

|   | 空表 | 插入89 | 插入18 | 插入49 | 插入58 | 插入69 |
|---|----|------|------|------|------|------|
| 0 |    |      |      | 49   | 49   | 49   |
| 1 |    |      |      |      | 58   | 58   |
| 2 |    |      |      |      |      | 69   |
| 3 |    |      |      |      |      |      |
| 4 |    |      |      |      |      |      |
| 5 |    |      |      |      |      |      |
| 6 |    |      |      |      |      |      |
| 7 |    |      |      |      |      |      |
| 8 |    |      | 18   | 18   | 18   | 18   |
| 9 |    | 89   | 89   | 89   | 89   | 89   |

图5-11 每次插入后使用线性探测得到的散列表

第一个冲突在插入键49时产生；它被放入下一个空闲地址，即地址0，该地址是开放的。键58先与18、89冲突，然后又与49冲突，试选三次之后才找到一个空单元。对69的冲突用类似的方法处理。只要表足够大，总能够找到一个自由单元，但是如此花费的时间是相当多的。更糟的是，即使表相对较空，这样占据的单元也会开始形成一些区块，其结果称为一次聚集（primary clustering），于是，散列到区块中的任何键都需要多次试选单元才能够解决冲突，然后该键才被添加到相应的区块中。

虽然这里不进行具体计算，但是可以证明，使用线性探测的预期探测次数对于插入和不成功的查找来说大约为 $\frac{1}{2}(1 + 1/(1 - \lambda)^2)$ ，而对于成功的查找来说则是 $\frac{1}{2}(1 + 1/(1 - \lambda))$ 。相关的一些计算多少有些复杂。从代码中容易看出，插入和不成功查找需要相同次数的探测。略加思考不难得出，成功查找应该比不成功查找平均花费较少的时间。

如果聚集不算是问题，那么对应的公式就很容易推导。假设有一个很大的表，并设每次探测都与前面的探测无关。对于随机冲突解决方法而言，这些假设是成立的，并且当 $\lambda$ 不是非常接近于1时也是合理的。首先，我们导出在一次不成功的查找中期望的探测次数，而这正是找到一个空单元所期望的探测次数。由于空单元所占的份额为 $1 - \lambda$ ，因此我们预计要探测的单元数是 $1/(1 - \lambda)$ 。一次成功的查找的探测次数等于该特定元素插入时所需要的探测次数。当插入一个元素时，可以看成是进行一次不成功查找的结果。因此，可以使用一次不成功查找的开销来计算一次成功查找的平均开销。

需要指出， $\lambda$ 从0到当前值之间变化，因此早期的插入操作开销较少，从而将平均开销拉低。例如，在上面的表中， $\lambda = 0.5$ ，访问18的开销是在18被插入时确定的，此时 $\lambda = 0.2$ 。由于18是插入到一个相对较空的表中，因此对它的访问应该比新近插入的元素（比如69）的访问更容易。我们可以通过使用积分计算插入时间平均值的方法来估计平均值，得到

$$I(\lambda) = \frac{1}{\lambda} \int_0^{\lambda} \frac{1}{1-x} dx = \frac{1}{\lambda} \ln \frac{1}{1-\lambda}$$

这些公式显然优于线性探测那些相应的公式。聚集不仅是理论上的问题，而且实际上也发生在具体的实现中。

图5-12把线性探测的性能（虚曲线）与其他更随机的冲突解决方案的期望性能做了比较。成功的查找用S标示，不成功查找和插入分别用U和I标记。



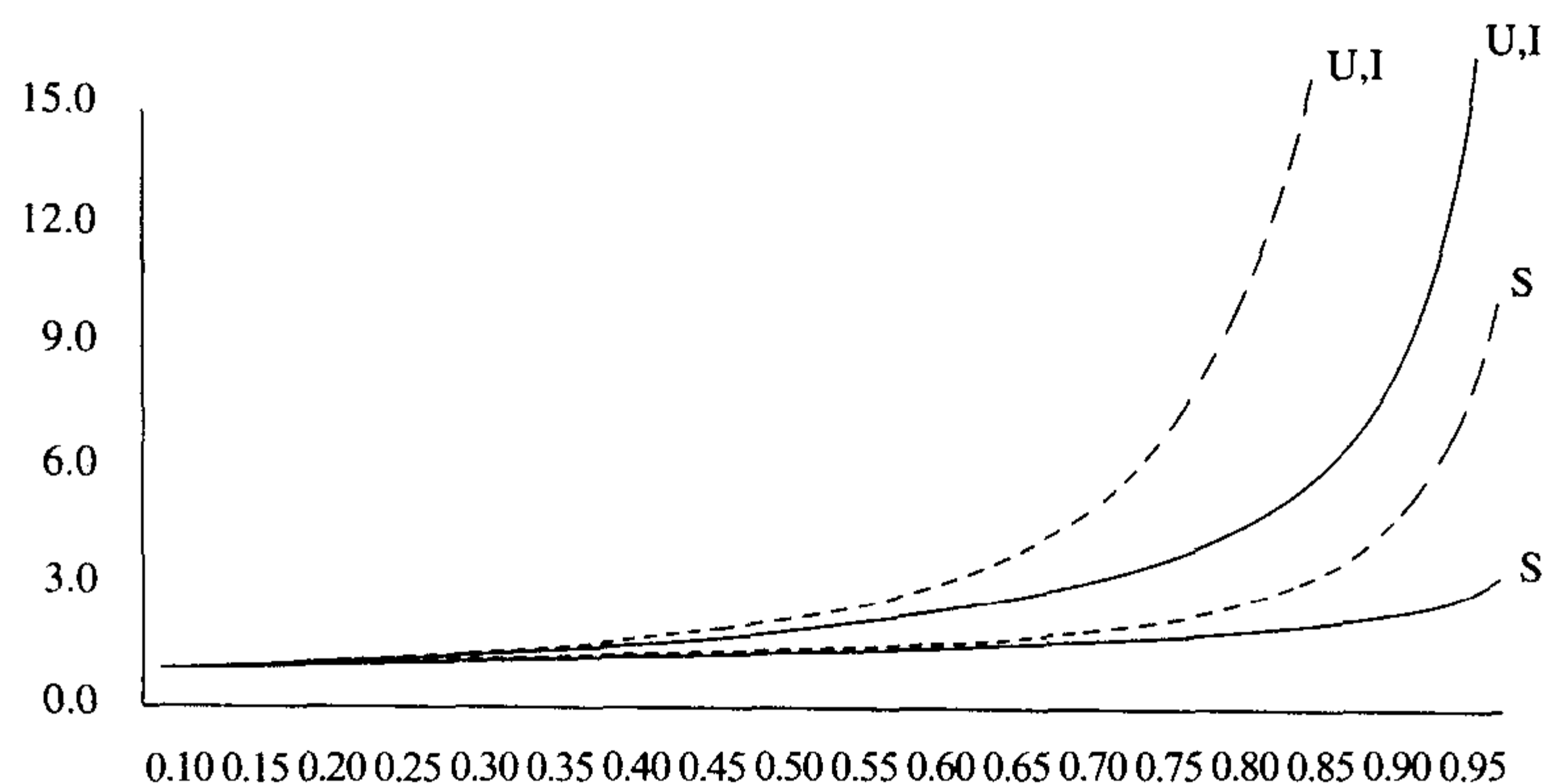


图5-12 对线性探测（虚线）和随机方法的装填因子画出的探测次数（S为成功查找，U为不成功查找，而I为插入）

194

如果 $\lambda = 0.75$ ，那么上面的公式指出在线性探测中一次插入预计需要8.5次探测。如果 $\lambda = 0.9$ ，则预计需要50次探测，这就不合理了。假如聚集不是问题，那么这可与对应装填因子的4次和10次探测相比。从这些公式看到，如果表可以有多于一半被填满的话，那么线性探测就不是个好办法。然而，如果 $\lambda = 0.5$ ，那么插入操作平均只需要2.5次探测，并且对于成功的查找平均只需要1.5次探测。

### 5.4.2 平方探测

平方探测是消除线性探测中一次聚集问题的冲突解决方法。平方探测就是冲突函数为二次函数的探测方法。流行的选择是 $f(i) = i^2$ ，图5-13显示了当输入与前面线性探测例子的输入相同时，使用该冲突函数所得到的散列表。

|   | 空表 | 插入89 | 插入18 | 插入49 | 插入58 | 插入69 |
|---|----|------|------|------|------|------|
| 0 |    |      |      | 49   | 49   | 49   |
| 1 |    |      |      |      |      |      |
| 2 |    |      |      |      | 58   | 58   |
| 3 |    |      |      |      |      | 69   |
| 4 |    |      |      |      |      |      |
| 5 |    |      |      |      |      |      |
| 6 |    |      |      |      |      |      |
| 7 |    |      |      |      |      |      |
| 8 |    |      | 18   | 18   | 18   | 18   |
| 9 |    | 89   | 89   | 89   | 89   | 89   |

图5-13 在每次插入后，利用平方探测得到的散列表

当49与89冲突时，其下一个位置为下一个单元，该单元是空的，因此49就被放在那里。此后，58在位置8处产生冲突，其后相邻的单元经探测得知发生了另外的冲突。下一个探测的单元在距位置8为 $2^2 = 4$ 远处，这个单元是个空单元。因此，键58就放在单元2处。对于键69，处理的过程也一样。

对于线性探测，让散列表近乎填满元素是个坏主意，因为此时表的性能会降低。对于平方探测，情况甚至更糟：一旦表被填满超过一半，若表的大小不是素数，那么甚至在表被填满一半之前，就不能保证找到空的单元了。这是因为最多只有表的一半可以用做解决冲突的备选位置。

现在就来证明, 如果表有一半是空的, 并且表的大小是素数, 那么可以保证总能够插入一个新的元素。

**定理5-1** 如果使用平方探测, 且表的大小是素数, 那么当表至少有一半是空的时候, 总能够插入一个新的元素。

**证明** 令表的大小  $TableSize$  是一个大于3的(奇)素数。我们证明, 前  $\lceil TableSize/2 \rceil$  个备选位置(包括初始位置  $h_0(x)$ )是互异的。其中的两个位置是  $h(x) + i^2 \pmod{TableSize}$  和  $h(x) + j^2 \pmod{TableSize}$ , 其中  $0 \leq i, j \leq \lfloor TableSize/2 \rfloor$ 。为推出矛盾, 假设这两个位置相同, 但  $i \neq j$ 。有 195

$$\begin{aligned} h(x) + i^2 &= h(x) + j^2 && \pmod{TableSize} \\ i^2 &= j^2 && \pmod{TableSize} \\ i^2 - j^2 &= 0 && \pmod{TableSize} \\ (i - j)(i + j) &= 0 && \pmod{TableSize} \end{aligned}$$

由于  $TableSize$  是素数, 因此, 或者  $(i - j)$  等于  $0 \pmod{TableSize}$ , 或者  $(i + j)$  等于  $0 \pmod{TableSize}$ 。既然  $i$  和  $j$  是互异的, 那么第一个选择是不可能的。但  $0 \leq i, j \leq \lfloor TableSize/2 \rfloor$ , 因此第二个选择也是不可能的。从而, 前  $\lceil TableSize/2 \rceil$  个备选位置是互异的。如果最多有  $\lfloor TableSize/2 \rfloor$  个位置被使用, 那么空单元总能够找到。 ■

如果哪怕表有比一半多一个的位置被填满, 那么插入都有可能失败(虽然这种可能性极小)。因此, 把它记住很重要。另外, 表的大小是素数也非常重要<sup>1</sup>。如果表的大小不是素数, 则备选单元的个数可能会锐减。例如, 若表的大小是16, 那么备选单元只能在距散列值1、4或9远处。

在探测散列表中标准的删除操作不能执行, 因为相应的单元可能已经引起过冲突, 元素绕过它存储在别处。例如, 如果我们删除89, 那么实际上所有剩下的 `find` 操作都将失败。因此, 探测散列表需要懒惰删除, 虽然在这种情况下实际上并不存在懒惰的意味。

实现探测散列表所需要的类接口在图5-14中给出。这里, 我们不使用链表数组, 而是使用散列表项单元数组。嵌套的类 `HashEntry` 存储在 `info` 成员中的一个项的状态, 这个状态可以是 `ACTIVE`、`EMPTY` 或 `DELETED`。在C++中, 这些常量可以声明为具有初始值的 `static const` 数据成员。这样, 在 `HashTable` 类中就可以得到:

```
static const int ACTIVE = 0;
static const int EMPTY = 1;
static const int DELETED = 2;
```

遗憾的是, 虽然这是C++标准要求的, 却不是所有的编译器都支持的。因此我们使用替代的枚举类型:

```
enum EntryType { ACTIVE, EMPTY, DELETED };
```

该类型也可以取得相同的效果。类型 `EntryType` 不再是 `int` 的, 而且 `ACTIVE`、`EMPTY` 和 `DELETED` 由编译器按顺序赋值。这个技巧很久以前就被C++的程序员用来声明整数类常量。例如:

```
enum { MAX_VALUE = 10 };
```

该表的构造(见图5-15)包括将每个单元的 `info` 成员设置为 `EMPTY`。在图5-16所示的 `contains(x)` 调用私有的成员函数 `isActive` 和 `finPos`。 `private` 成员函数 `findPos` 解决冲突问

1. 如果表的大小是形如  $4k + 3$  的素数, 且使用的平方冲突解决方法为  $f(i) = \pm i^2$ , 那么整个表均可被探测到。其代价则是例程要略微复杂。

题。在insert例程里，我们确保散列表至少有表中元素的两倍大，这样平方探测解法总可以实现。在图5-16的实现中，删除的元素依然存在于表中，只是标记删除的次数。这可能引起一些问题，因为该表可能提前过满。我们现在就来讨论该问题。

197

```

1  template <typename HashedObj>
2  class HashTable
3  {
4  public:
5      explicit HashTable( int size = 101 );
6
7      bool contains( const HashedObj & x ) const;
8
9      void makeEmpty( );
10     bool insert( const HashedObj & x );
11     bool remove( const HashedObj & x );
12
13     enum EntryType { ACTIVE, EMPTY, DELETED };
14
15 private:
16     struct HashEntry
17     {
18         HashedObj element;
19         EntryType info;
20
21         HashEntry( const HashedObj & e = HashedObj( ), EntryType i = EMPTY )
22             : element( e ), info( i ) { }
23     };
24
25     vector<HashEntry> array;
26     int currentSize;
27
28     bool isActive( int currentPos ) const;
29     int findPos( const HashedObj & x ) const;
30     void rehash( );
31     int myhash( const HashedObj & x ) const;
32 };

```

图5-14 使用探测策略的散列表的类接口，包括嵌套的HashEntry类

```

1  explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
2  { makeEmpty( ); }
3
4  void makeEmpty( )
5  {
6      currentSize = 0;
7      for( int i = 0; i < array.size( ); i++ )
8          array[ i ].info = EMPTY;
9  }

```

图5-15 初始化平方探测散列表的例程

第12行至第15行给出了进行平方探测的快速方法。由平方解法函数的定义可知， $f(i) = f(i-1) + 2i - 1$ ，因此，下一个要探测的单元的位置由上一次探测所使用的距离加上自增2后的偏移距离所得的和来确定。如果新的定位越过数组，那么可以通过减去TableSize把它拉回到数组范围内。这比通常的方法要快，因为它避免了看似需要的乘法和除法。重要警告：第9行和第10行的测试顺序很重要，切勿交换！

198

最后的例程是插入。如分离链接散列法那样，若x已经存在，就什么都不做。其他工作只是

简单的修改。若x不存在，那么我们就把要插入的元素x放在findPos例程指出的地方。程序代码在图5-17中给出。如果装填因子超过0.5，那么表是满的，于是将该散列表扩大。这称为再散列(rehashing)，将在5.5节进行讨论。图5-17也给出了remove例程。

```

1 bool contains( const HashedObj & x ) const
2   { return isActive( findPos( x ) ); }
3
4 int findPos( const HashedObj & x ) const
5 {
6     int offset = 1;
7     int currentPos = myhash( x );
8
9     while( array[ currentPos ].info != EMPTY &&
10          array[ currentPos ].element != x )
11     {
12         currentPos += offset; // Compute ith probe
13         offset += 2;
14         if( currentPos >= array.size( ) )
15             currentPos -= array.size( );
16     }
17
18     return currentPos;
19 }
20
21 bool isActive( int currentPos ) const
22     { return array[ currentPos ].info == ACTIVE; }

```

图5-16 使用平方探测进行散列的contains例程（和它的private助手）

```

1 bool insert( const HashedObj & x )
2 {
3     // Insert x as active
4     int currentPos = findPos( x );
5     if( isActive( currentPos ) )
6         return false;
7
8     array[ currentPos ] = HashEntry( x, ACTIVE );
9
10    // Rehash; see Section 5.5
11    if( ++currentSize > array.size( ) / 2 )
12        rehash( );
13
14    return true;
15 }
16
17 bool remove( const HashedObj & x )
18 {
19     int currentPos = findPos( x );
20     if( !isActive( currentPos ) )
21         return false;
22
23     array[ currentPos ].info = DELETED;
24     return true;
25 }

```

图5-17 使用平方探测的散列表的insert和remove例程

虽然平方探测排除了一次聚集，但是散列到同一位置上的那些元素将探测相同的备选单元。这称为二次聚集(secondary clustering)。二次聚集是理论上的一个小缺憾。模拟结果指出，对每



次查找，它一般要引起另外的少于一半的探测。下面的技术可以消除这个缺憾，但是，这是以计算额外的散列函数为代价的。

5.4.3 双散列

我们要讨论的最后一个冲突解决方法是双散列（double hashing）。对于双散列，一种流行的选择是 $f(i) = i \cdot hash_2(x)$ 。这个公式是说，将第二个散列函数应用到 $x$ 并在距离 $hash_2(x), 2hash_2(x), \dots$ 等处探测。 $hash_2(x)$ 选择得不好将会非常糟糕。例如，若把99插入到前面例子的输入中去，则通常的选择 $hash_2(x) = x \bmod 9$ 将不起作用。因此，函数的运算结果不可以是0。另外，保证所有的单元都能被探测到（在下面的例子中这是不可能的，因为表的大小不是素数）也是很重要的。诸如 $hash_2(x) = R - (x \bmod R)$ 这样的函数将起到良好的作用，其中 $R$ 为小于 $TableSize$ 的素数。如果我们选择 $R = 7$ ，则图5-18显示了插入与前面相同的键的结果。

199

|   | 空表 | 插入89 | 插入18 | 插入49 | 插入58 | 插入69 |
|---|----|------|------|------|------|------|
| 0 |    |      |      |      |      | 69   |
| 1 |    |      |      |      |      |      |
| 2 |    |      |      |      | 58   | 58   |
| 3 |    |      |      |      |      |      |
| 4 |    |      |      |      |      |      |
| 5 |    |      |      |      |      |      |
| 6 |    |      |      | 49   | 49   | 49   |
| 7 |    |      |      |      |      |      |
| 8 |    |      | 18   | 18   | 18   | 18   |
| 9 |    | 89   | 89   | 89   | 89   | 89   |

图5-18 使用双散列方法在每次插入后的散列表

第一个冲突发生在插入49的时候。 $hash_2(49) = 7 - 0 = 7$ ，故49被插入到位置6。 $hash_2(58) = 7 - 2 = 5$ ，于是58被插入到位置3。最后，69产生冲突，从而被插入到距离为 $hash_2(69) = 7 - 6 = 1$ 的地方。如果我们试图将60插入到位置0处，那么就会产生一个冲突。由于 $hash_2(60) = 7 - 4 = 3$ ，因此我们尝试位置3、6、9，然后是2，直到找出一个空的单元。一般有可能发现某个坏情形，不过这样的坏情形并不多。

前面已经提到，上面实例中散列表的大小不是素数。这么做是为了计算散列函数时方便，但是，有必要了解在使用双散列时为什么保证表的大小为素数是重要的。如果想要把23插入到表中，那么它就会与58发生冲突。由于 $hash_2(23) = 7 - 2 = 5$ ，且该表大小是10，因此实际上只有一个备选位置，而这个位置已经使用了。因此，如果表的大小不是素数，那么备选单元就有可能提前用完。然而，如果双散列正确实现，则模拟表明，预期的探测次数几乎和随机冲突解决方法的情形相同。这使得双散列理论上很有吸引力。不过，平方探测不需要使用第二个散列函数，从而在实践中可能更简单并且更快。当键为字符串时，尤其如此，因为其散列函数的计算很耗时。

5.5 再散列

对于使用平方探测的开放定址散列法，如果表的元素填得太满，那么操作的运行时间将开始消耗过长，且插入操作可能失败。这可能发生在有太多的删除和插入混合的场合。此时，一个解决方法是建立另外一个大约两倍大的表（而且使用一个相关的新散列函数），扫描整个原始散列

200

表，计算每个（未删除的）元素的新散列值并将其插入到新表中。

例如，设将元素13、15、24和6插入到大小为7的线性探测散列表中。散列函数是 $h(x) = x \bmod 7$ 。插入后得到的散列表如图5-19所示。

|   |    |
|---|----|
| 0 | 6  |
| 1 | 15 |
| 2 |    |
| 3 | 24 |
| 4 |    |
| 5 |    |
| 6 | 13 |

图5-19 使用线性探测插入13，15，6，24的散列表

如果将23插入表中，那么图5-20中插入后的表将有超过70%的单元是满的。因为表太满，所以需要建立一个新的表。新表的大小为17，这是因为17是原表大小两倍后的第一个素数。新的散列函数为 $h(x) = x \bmod 17$ 。扫描原来的表，并将元素6、15、23、24和13插入到新表中。最后得到的表见图5-21。

|   |    |
|---|----|
| 0 | 6  |
| 1 | 15 |
| 2 | 23 |
| 3 | 24 |
| 4 |    |
| 5 |    |
| 6 | 13 |

图5-20 使用线性探测插入23后的散列表

|    |    |
|----|----|
| 0  |    |
| 1  |    |
| 2  |    |
| 3  |    |
| 4  |    |
| 5  |    |
| 6  | 6  |
| 7  | 23 |
| 8  | 24 |
| 9  |    |
| 10 |    |
| 11 |    |
| 12 |    |
| 13 | 13 |
| 14 |    |
| 15 | 15 |
| 16 |    |

图5-21 再散列后的散列表

整个操作称为**再散列**（rehashing）。这显然是一种非常昂贵的操作；其运行时间为 $O(N)$ ，因为要有 $N$ 个元素要再散列而且表的大小约为 $2N$ ，不过，因为不是经常发生，所以实际效果并没有这么差。特别是，在最后的再散列之前必然已经存在 $N/2$ 次插入，因此添加到每个插入上的花费基本上是一个常数开销<sup>1</sup>。如果这种数据结构是程序的一部分，那么其影响是不明显的。另一方面，

1. 这也是新表的大小要是旧表的两倍的原因。

如果再散列作为交互系统的一部分运行，那么其插入引起再散列的用户将会感到速度减慢。

再散列可以用平方探测以多种方法实现。一种做法是只要表满到一半就再散列。另一种极端的方法是只有当插入失败时才再散列。第三种方法即**途中**（middle-of-the-road）策略：当表到达某一个装填因子时进行再散列。由于随着装填因子的增加，表的性能的确在下降，因此，以好的截止点实现的第三种策略，可能是最好的策略。

在分离链接散列表中的再散列是相似的。图5-22提供了一个分离链接再散列的实现。可以看出，再散列是很容易实现的。

```

1  /**
2   * Rehashing for quadratic probing hash table.
3   */
4  void rehash( )
5  {
6      vector<HashEntry> oldArray = array;
7
8      // Create new double-sized, empty table
9      array.resize( nextPrime( 2 * oldArray.size( ) ) );
10     for( int j = 0; j < array.size( ); j++ )
11         array[ j ].info = EMPTY;
12
13     // Copy table over
14     currentSize = 0;
15     for( int i = 0; i < oldArray.size( ); i++ )
16         if( oldArray[ i ].info == ACTIVE )
17             insert( oldArray[ i ].element );
18 }
19
20 /**
21 * Rehashing for separate chaining hash table.
22 */
23 void rehash( )
24 {
25     vector<list<HashedObj> > oldLists = theLists;
26
27     // Create new double-sized, empty table
28     theLists.resize( nextPrime( 2 * theLists.size( ) ) );
29     for( int j = 0; j < theLists.size( ); j++ )
30         theLists[ j ].clear( );
31
32     // Copy table over
33     currentSize = 0;
34     for( int i = 0; i < oldLists.size( ); i++ )
35     {
36         list<HashedObj>::iterator itr = oldLists[ i ].begin( );
37         while( itr != oldLists[ i ].end( ) )
38             insert( *itr++ );
39     }
40 }

```

图5-22 对分离链接散列表和探测散列表的再散列

## 5.6 标准库中的散列表

标准库中不包括set和map的散列表实现。但是，许多的编译器提供具有与set和map类相同的成员函数的hash\_set和hash\_map。

要使用hash\_set和hash\_map，就必须有相应的包含指令，而且，可能也需要相应的命名空间。这两者都是和编译器相关的。接下来还必须提供相应的类型参数来说明hash\_set和hash\_map。对于hash\_map，这些类型参数包括键的类型、值的类型、散列函数（返回无符号整数）和一个相等性操作符。遗憾的是，至于键和值的类型参数如何表示还是编译器相关的。

下一次C++的较大修订将不可避免地包括这些hash\_set和hash\_map中的一个。

5.7 可扩散列

本章最后的论题处理数据量太大以至于装不进主存的情况。正如在第4章见到的，此时主要考虑的是检索数据所需的磁盘存取次数。

与前面一样，假设在任意时刻都有N个记录要存储；N的值随时间而变化。此外，最多可把M个记录放入一个磁盘区块。本节中设M=4。

如果使用探测散列或分离链接散列，那么主要的问题在于，即使是理想分布的散列表，在一次查找操作中，冲突也可能引起对多个区块的访问。不仅如此，当表变得过满的时候，必须执行代价巨大的再散列这一步，它需要O(N)的磁盘访问。

一种聪明的选择称为可扩散列（extendible hashing），它允许用两次磁盘访问执行一次查找。插入操作也需要很少的磁盘访问。

回忆第4章，B树的深度为O(log<sub>M/2</sub> N)。随着M的增加，B树的深度降低。理论上可以选择M很大，使得B树的深度为1。此时，在第一次以后的任何查找都将花费一次磁盘访问，因为根结点可能存在主存中。这种方法的问题在于分支系数（branching factor）太高，以至于为了确定数据在哪片树叶上要进行大量的处理工作。如果运行这一步的时间可以缩减，那么我们就将有一个实际的方案。这正是可扩散列使用的策略。

现在假设，数据由几个6位整数组成。图5-23显示了这些数据的可扩散列格式。这里“树”的根含有4个指针，它们由这些数据的前两个位确定。每片树叶至多有M=4个元素。碰巧这里每片树叶中数据的前两个位都是相同的；这由圆括号内的数指出。为了更正式，用D代表根所使用的位数，有时称其为目录（directory）。于是，目录中的项数为2<sup>D</sup>。d<sub>L</sub>为树叶L所有元素共有的最高位的位数。d<sub>L</sub>依赖于特定的树叶，因此d<sub>L</sub>≤D。

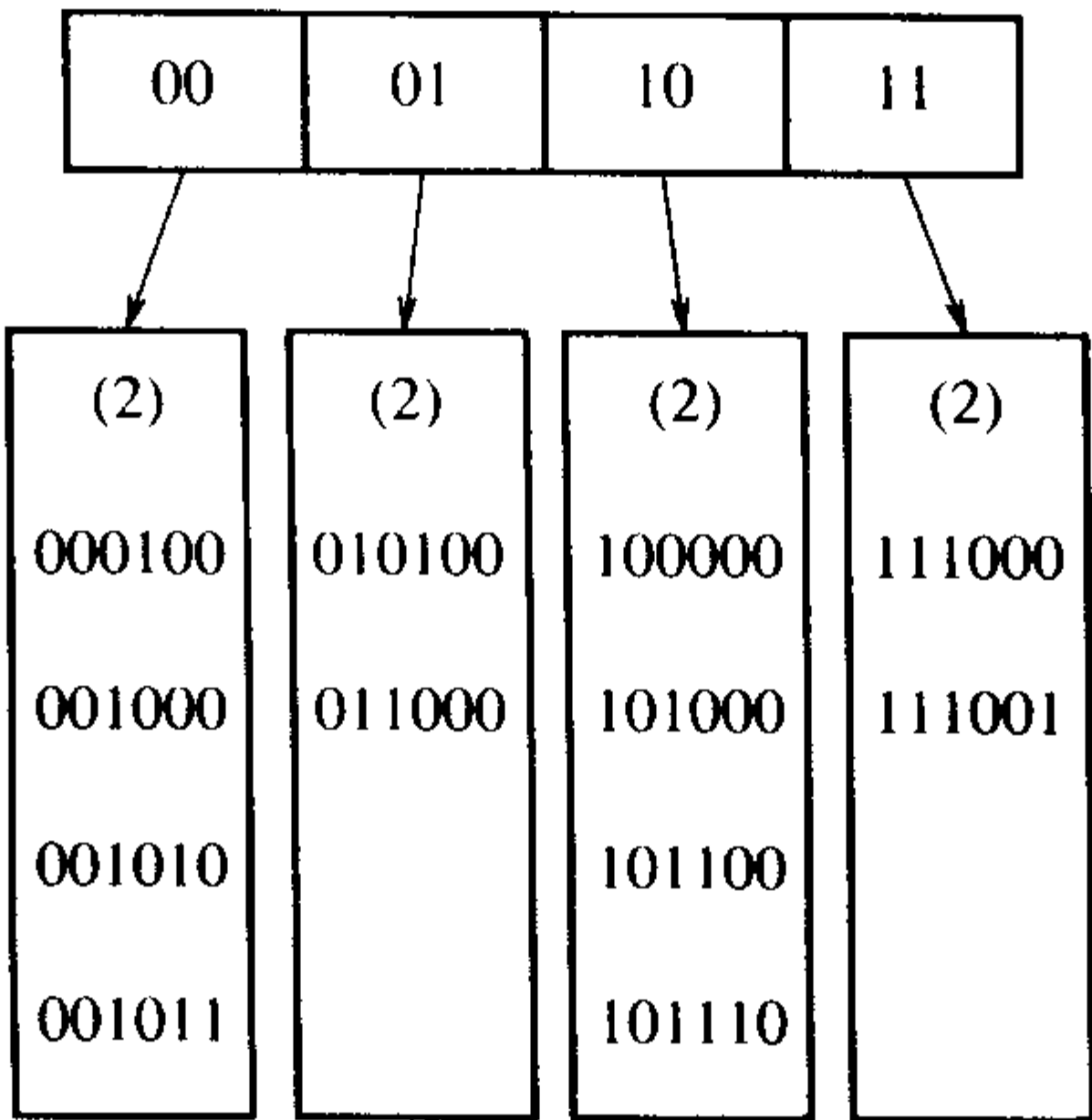


图5-23 可扩散列：原始数据

设欲插入键100100。它将进入第三片树叶，但是第三片树叶已经满了，没有空间存放它。因



此将这片树叶分裂成两片树叶，这由前三个位确定。同时需要将目录的大小增加到3。图5-24反映了这些变化。

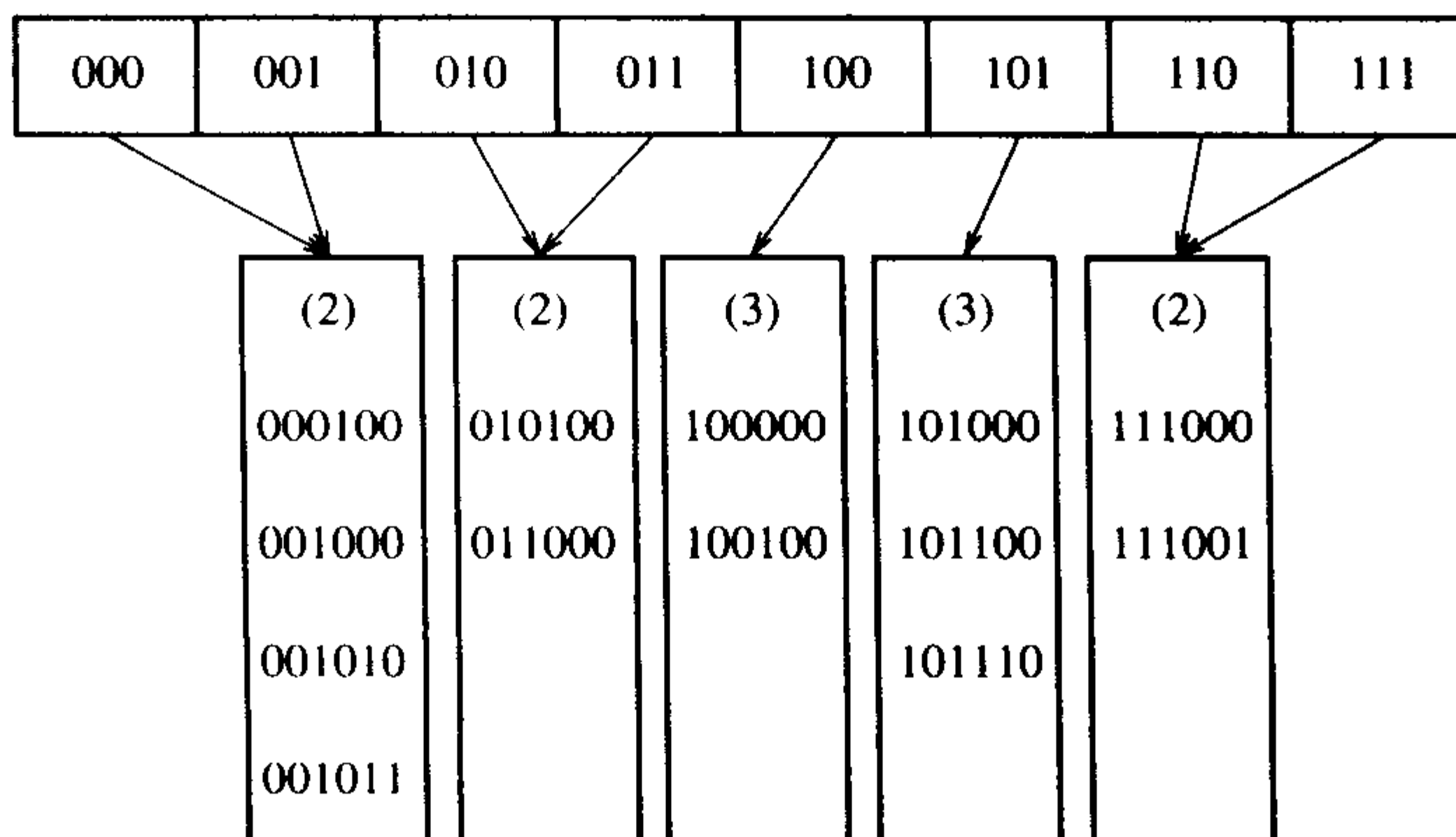


图5-24 可扩散列：在100100插入及目录分裂后

205

注意，所有未被分裂的树叶现在各由两个相邻目录项所指。因此，虽然整个目录被重写，但是实际上其他树叶并没有访问到。

如果现在插入键000000，那么第一片树叶就要被分裂，生成 $d_L = 3$ 的两片树叶。由于 $D = 3$ ，故在目录中所做的唯一变化是000和001两个指针的更新，见图5-25。

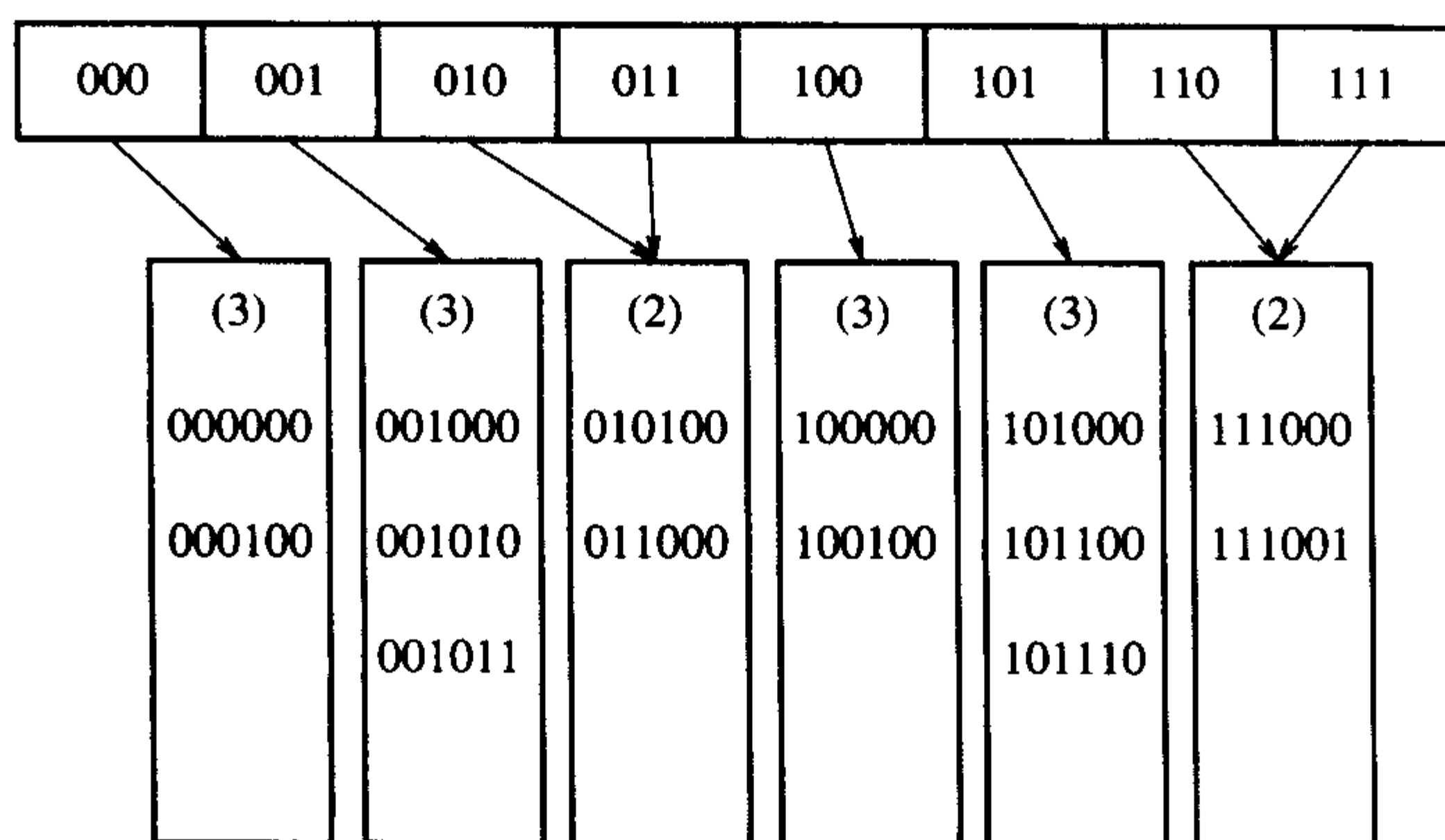


图5-25 可扩散列：在000000插入及树叶分裂后

这个非常简单的方法提供了对大型数据库insert操作和查找操作的快速存取时间。另外，还有一些重要细节我们尚未考虑。

首先，当一片树叶的元素有多于 $D + 1$ 个前导位时，有可能需要多个目录分裂。例如，从原先的例子开始， $D = 2$ ，如果插入111010、111011，并在最后插入111100，那么目录大小必须增加到4以区分5个键。这是一个容易考虑到的细节，但是千万不要忘记它。其次，存在重复键(duplicate keys)的可能性；若存在多于 $M$ 个重复键，则该算法根本无法工作。此时，需要做出某些其他的安排。

上述可能性指出，这些位完全随机是相当重要的，这可以通过把这些键散列到合理的长整数（因而是名字）来完成。

最后，我们介绍可扩散列的某些性能，这些性能是经过非常困难的分析后得到的。这些结果

基于合理的假设：位模式（bit pattern）是均匀分布的。

树叶的期望个数为 $(N/M)\log_2 e$ 。因此，平均树叶满的程度为 $\ln 2 = 0.69$ 。这和B树是一样的，其实这完全不奇怪，因为对于两种数据结构都是当第 $(M+1)$ 项被添加进来时，一些新的结点就建立起来。

更惊奇的结果是，目录的期望大小（换句话说即 $2^D$ ）为 $O(N^{1+1/M}/M)$ 。如果 $M$ 很小，那么目录可能过分地大。在这种情况下，我们可以让树叶包含指向记录的指针而不是实际的记录，这样可以增加 $M$ 的值。为了维持更小的目录，可以把第二次磁盘访问添加到每个查找操作中去。如果目录太大装不进主存，那么第二次磁盘访问无论如何也是需要的。

206

## 小结

散列表可以用来以常数平均时间实现insert和contains操作。当使用散列表时，注意诸如装填因子这样的细节是特别重要的，否则时间界将不再有效。当键不是短字符串或整数时，仔细选择散列函数也是很重要的。

对于分离链接散列法，虽然装填因子不大时性能并不明显降低，但装填因子还是应该接近于1。对于探测散列，除非完全不可避免，否则装填因子不应该超过0.5。如果使用线性探测，那么性能随着装填因子接近于1而急速下降。再散列运算可以通过使表增长（和收缩）来实现，这样可以保持合理的装填因子。对于空间紧缺并且不可能声明巨大散列表的情况，这是很重要的。

二叉查找树也可以用来实现insert和contains操作。虽然平均时间界为 $O(\log N)$ ，但是二叉查找树也支持那些需要排序的例程，从而功能更强大。使用散列表不可能找出最小元素。除非准确知道一个字符串，否则散列表也不可能有效地查找它。二叉查找树可以迅速找到一定范围内的所有项；散列表却做不到。不仅如此，因为查找树不需要乘法和除法， $O(\log N)$ 这个时间界也不必比 $O(1)$ 大那么多。

另一方面，散列的最坏情形一般来自于实现错误，而有序的输入却可能使二叉树运行得很差。平衡查找树实现的代价相当高，因此，如果不需要排序的信息或者不确定输入是否已经排序，那么就应该选择散列这种数据结构。

散列的应用很广。编译器使用散列表跟踪源代码中声明的变量，这种数据结构叫作符号表（symbol table）。散列表是这种问题的理想选择。标识符一般都不长，因此散列函数能够迅速完成运算。此外，按字母顺序排列变量通常也是不必要的。

散列表适用于任何其结点有实名而不是数字名的图论问题。这里，当输入被读入的时候，顶点则按照它们出现的顺序从1开始指定为一些整数。再有，输入很可能有一组一组按字母顺序排列的项。例如，顶点可以是计算机。此时，如果一个特定的计算中心把它的计算机列表成ibm1, ibm2, ibm3, …，那么，若使用查找树则在效率方面可能会有戏剧性的效果。

散列表的第三种常见的用途是在为游戏编制的程序中。当程序搜索游戏的不同的运动路径时，它通过计算基于位置的散列函数而跟踪一些已知的位置（并把对于该位置的移动存储起来）。如果同样的位置再次出现，程序通常通过简单的移动变换来避免昂贵的重复计算。游戏程序的这种一般特点叫作置换表（transposition table）。

散列的另一个用途是在线拼写检查程序。如果拼写检查程序的主要功能是检查拼写错误（而非纠正错误），那么可以预先将整个词典进行散列，这样就可以在常数时间内检查单词拼写。散列表很适合这项工作，因为以字母顺序排列单词并不重要；而以它们在文件中出现的顺序显示错误拼写当然也是可以接受的。

207

我们通过返回到第1章的字谜问题来结束这一章。如果使用第1章中描述的第二个算法，并且假设最大的单词的长度是某个小常数，那么读入包含 $W$ 个单词的词典并把它放入散列表的时间是 $O(W)$ 。这个时间的大小很可能由磁盘I/O而不是由那些散列例程来支配的。算法的其余部分将对每一个四元组（行，列，方向，字符数）测试一个单词是否出现。由于每次查询时间为 $O(1)$ ，而且只存在常数个方向（8）和每个单词的字符，因此这一阶段的运行时间为 $O(R \cdot C)$ 。总的运行时间是 $O(R \cdot C + W)$ ，这是对原始 $O(R \cdot C \cdot W)$ 的明显的改进。还可以做进一步的优化，它能够降低实际的运行时间；这将在练习中描述。

## 练习

- 5.1 给定输入{4371, 1323, 6173, 4199, 4344, 9679, 1989}和散列函数 $h(x) = x \pmod{10}$ ，指出下列结果：
  - a. 分离链接散列表。
  - b. 使用线性探测的散列表。
  - c. 使用平方探测的散列表。
  - d. 第二个散列函数为 $h_2(x) = 7 - (x \pmod{7})$ 的散列表。
- 5.2 指出将练习5.1中的散列表再散列的结果。
- 5.3 编写一个程序，计算使用线性探测、平方探测以及双散列的长随机插入序列所需要的冲突次数。
- 5.4 在分离链接散列表中进行大量的删除可能造成表非常稀疏，浪费空间。在这种情况下，我们可以再散列一个表，为原表的一半大。当元素的个数大于表的大小的两倍时，就再散列到一个更大的表。那么当表应该有多么稀疏时，才能再散列到一个更小的表？
- 5.5 平方探测的isEmpty例程没有写出。你能通过返回表达式`currentSize == 0`实现它吗？
- 5.6 在平方探测散列表中，设我们把一个新元素插入到搜索路径上第一个非活动的单元而不是把它插入到由findPos指定的位置（这样，有可能回收一个标记“deleted”的单元，潜在地节省了空间）。
  - a. 使用上述结果重新编写插入算法。通过使用一个附加变量让findPos保持它遇到的第一个非活动单元的位置来完成重写的工作。
  - b. 解释使得重写的算法快于原来算法的环境。重写的算法有可能会慢吗？
- 5.7 图5-4中的散列函数在for循环中对key.length()进行重复调用。每次进入循环以前对它进行一次计算值得吗？
- 5.8 各种冲突解决方法的优点和缺点是什么？
- 5.9 假设为减弱二次聚集的影响，使用冲突解决函数 $f(i) = i \cdot r(\text{hash}(x))$ ，这里 $\text{hash}(x)$ 是32位的散列值（尚未分配至适当的数组索引），而 $r(y) = \lfloor 48271 y \pmod{2^{31}-1} \rfloor \pmod{\text{TableSize}}$ 。（10.4.1节描述了一个执行这个运算且不溢出的方法，但是在这个例子中不太可能溢出。）解释为什么这个策略有助于避免二次聚集，并将这个策略与双散列和平方探测相比较。
- 5.10 再散列需要对散列表中的所有项重新计算散列函数。由于计算散列函数的代价很昂贵，设每个对象提供自己的散列成员函数，并且每个对象都在一个附加的数据成员里存储第一次运行其散列函数的计算结果。说明这样的程序构架如何应用于图5-8的Employee类。并解释在什么情况下在每个Employee中存储的散列值维持有效。
- 5.11 编写一个程序来用如下的策略实现两个稀疏多项式（sparse polynomial） $P_1$ 和 $P_2$ 的相乘。其中 $P_1$ 和 $P_2$ 的大小分别为 $M$ 和 $N$ 。每个多项式表示成一个包含系数和幂的表。将 $P_1$ 中的每一项与 $P_2$ 中的每一项相乘，共有 $MN$ 次操作。一个办法是将这些项排序，然后合并同类项。这需要排序 $MN$ 个记录，其代价是昂贵的，尤其是在小内存的环境里。另一种方案是在计算的时候就合并同类项，然后再对结果排序。
  - a. 写一个程序来实现第二种方案。



- b. 如果多项式的输出有大约 $O(M+N)$ 项, 那么每种方法的运行时间是多少?
- \*5.12 描述一个避免初始化散列表的过程 (以内存消耗为代价)。
- 5.13 设要找出在长输入字符串 $A_1A_2\cdots A_N$ 中字符串 $P_1P_2\cdots P_k$ 第一次出现的位置。可以通过散列模式字符串得到一个散列值 $H_P$ , 并将该值与从 $A_1A_2\cdots A_k, A_2A_3\cdots A_{k+1}, A_3A_4\cdots A_{k+2}$ , 等等, 直到 $A_{N-k+1}A_{N-k+2}\cdots A_N$ 形成的散列值比较来解决这个问题。如果我们得到散列值的一个匹配, 那么再一个字符一个字符地对字符串进行比较以检验这个匹配。如果串实际上确实匹配, 那么返回其 (在 $A$ 中的) 位置, 否则就继续进行查找 (匹配失败这种情况不大可能发生)。
- \*a. 证明如果 $A_iA_{i+1}\cdots A_{i+k-1}$ 的散列值已知, 那么 $A_{i+1}A_{i+2}\cdots A_{i+k}$ 的散列值可以在常数时间内算出。
  - b. 证明运行时间为 $O(k+N)$ 加上检验假匹配所耗费的时间。
  - \*c. 证明假匹配的期望次数是微不足道的。
  - d. 编写一个程序实现该算法。
  - \*e. 描述一个算法, 其最坏情形的运行时间为 $O(k+N)$ 。
  - \*\*f. 描述一个算法, 其平均运行时间为 $O(N/k)$ 。
- 5.14 一个 (老式的) BASIC程序由一系列按递增顺序编号的语句组成。控制是通过使用`goto`或`gosub`后加一个语句编号实现的。编写一个程序读进合法的BASIC程序并给语句重新编号, 使得第一句在序号 $F$ 处开始并且每一个语句的序号比前一语句的序号高 $D$ 。可以假设 $N$ 条语句的一个上限, 但是输入的语句序号可以是32位那么大的整数。你的程序必须以线性时间运行。
- 5.15
- a. 利用本章末尾描述的算法实现字谜程序。
  - b. 通过存储每一个单词 $W$ 以及 $W$ 的所有前缀, 可以大大加快运行速度 (如果 $W$ 的一个前缀刚好是词典中的一个单词, 那么就把它作为实际的单词来存储)。虽然这看起来极大地增加了散列表的大小, 但实际上并非如此, 因为许多单词都有相同的前缀。当以某个特定的方向执行一次扫描的时候, 如果被查找的单词前缀都不在散列表中, 那么在这个方向上的扫描可以及早终止。利用这种想法编写一个改进的程序来解决字谜游戏问题。
  - c. 如果可以牺牲散列表ADT的严肃性, 那么可以在b部分使程序加速: 例如, 如果刚刚计算出“excel”的散列函数, 那么就不必再从头开始计算“excels”的散列函数。调整散列函数使得它能够利用前面的计算。
  - d. 在第2章我们建议使用二分搜索。把使用前缀的想法结合到你的二分搜索算法中。修改工作应该很简单。哪个算法更快?
- 5.16 在某些假设下, 向带有二次聚集的散列表进行的一次插入操作的期望代价为 $1/(1-\lambda) - \lambda - \ln(1-\lambda)$ 。不过, 这个公式对于平方探测不精确。我们假设它是准确的, 确定:
- a. 一次不成功查找的期望代价。
  - b. 一次成功查找的期望代价。
- 5.17 实现支持`insert`和`lookup`操作的一般的Map。该实现将存储 (键, 定义) 对的散列表。你将通过提供一个键来`lookup`一个定义。图5-26提供Map的说明 (去掉某些细节)。
- 5.18 通过使用散列表实现一个拼写检查程序。设词典有两个来源: 一本现有的大词典以及由一本个人词典组成的第二个文件。输出所有拼错的单词和这些单词出现的行号。再有, 对于每个错拼的单词, 列出应用下列规则在词典中能够得到的任意的单词:
- a. 添加一个字母。
  - b. 去掉一个字母。
  - c. 交换两个相邻的字母。
- 5.19 指出将键10111101、00000010、10011011、10111110、01111111、01010001、10010110、00001011、11001111、10011110、11011011、00101011、01100001、11110000、01101111插入到一个空的初始可扩散列数据结构中的结果, 其中 $M=4$ 。



```

1  template <typename HashedObj, typename Object>
2  class Pair
3  {
4      HashedObj key;
5      Object    def;
6      // Appropriate Constructors, etc.
7  };
8
9  template <typename HashedObj, typename Object>
10 class Dictionary
11 {
12     public:
13         Dictionary( );
14
15         void insert( const HashedObj & key, const Object & definition );
16         const Object & lookup( const HashedObj & key ) const;
17         bool isEmpty( ) const;
18         void makeEmpty( );
19
20     private:
21         HashTable<Pair<HashedObj, Object> > items;
22 };

```

图5-26 练习5.17的词典构架

5.20 编写一个程序实现可扩散列。如果表小到足可装入内存，那么它的性能与分离链接法和开放定址散列法相比如何？

210

## 参考文献

尽管散列的简单是显而易见，但是对它的很多分析还是相当困难的，而且仍然留有许多尚未解决的问题。也还存在诸多有趣的理论问题，它们一般试图使得散列的最坏情形尽可能不出现。

散列的早期论文是[16]。关于这方面丰富的信息，包括对使用线性探测的散列的分析，可以在[11]中找到。[14]是对该课题极好的综述；[15]包含选择散列函数的一些建议以及一些要注意的陷阱。对于本章描述的所有方法的精确分析和模拟结果可以在[8]中找到。

对双散列的分析见于[9]和[13]。另外一种冲突解决方案是接合散列（coalesced hashing），[17]对此做了描述。Yao[19]业已证明，关于一次成功查找的开销，均匀散列（uniform hashing）是最优的，在这种散列中不存在聚集。

如果输入的键事先已知，那么完美散列函数存在，它不产生冲突，见[2]和[7]。某些更复杂的散列方案出现在[3]和[4]中，对于这些方案，最坏的情形并不依赖于特定的输入，而是依赖于算法所选择的随机数。

可扩散列出自[5]，分析见于[6]和[18]。

211

练习5.13（a~d）取自[10]，（e）部分取自[12]，而（f）部分取自[1]。

1. R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, 20 (1977), 762-772.
2. J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Sciences*, 18 (1979), 143-154.
3. M. Dietzfelbinger, A. R. Karlin, K. Melhorn, E Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds," *SIAM Journal on Computing*, 23 (1994), 738-761.
4. R.J. Enbody and H. C. Du, "Dynamic Hashing Schemes," *Computing Surveys*, 20 (1988), 85-113.
5. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing—A Fast Access Method

- for Dynamic Files,” *ACM Transactions on Database Systems*, 4 (1979), 315-344.
6. P. Flajolet, “On the Performance Evaluation of Extendible Hashing and Trie Searching,” *Acta Informatica*, 20 (1983), 345-369.
  7. M. L. Fredman, J. Komlos, and E. Szemerédi, “Storing a Sparse Table with  $O(1)$  Worst Case Access Time,” *Journal of the ACM*, 31 (1984), 538-544.
  8. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2nd ed., Addison-Wesley, Reading, Mass., 1991.
  9. L.J. Guibas and E. Szemerédi, “The Analysis of Double Hashing,” *Journal of Computer and System Sciences*, 16 (1978), 226-274.
  10. R. M. Karp and M. O. Rabin, “Efficient Randomized Pattern-Matching Algorithms,” *Aiken Computer Laboratory Report TR-31-81*, Harvard University, Cambridge, Mass., 1981.
  11. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., Addison-Wesley, Reading, Mass., 1998.
  12. D. E. Knuth, J. H. Morris, and V R. Pratt, “Fast Pattern Matching in Strings,” *SIAM Journal on Computing*, 6 (1977), 323-350.
  13. G. Lueker and M. Molodowitch, “More Analysis of Double Hashing,” *Proceedings of the Twentieth ACM Symposium on Theory of Computing* (1988), 354-359.
  14. W. D. Maurer and T. G. Lewis, “Hash Table Methods,” *Computing Surveys*, 7 (1975), 5-20.
  15. B.J. McKenzie, R. Harries, and T. Bell, “Selecting a Hashing Algorithm,” *Software—Practice and Experience*, 20 (1990), 209-224.
  16. W. W. Peterson, “Addressing for Random Access Storage,” *IBM Journal of Research and Development*, 1 (1957), 130-146.
  17. J. S. Vitter, “Implementations for Coalesced Hashing,” *Communications of the ACM*, 25 (1982), 911-926.
  18. A. C. Yao, “A Note on the Analysis of Extendible Hashing,” *Information Processing Letters*, 11 (1980), 84-86.
  19. A. C. Yao, “Uniform Hashing Is Optimal,” *Journal of the ACM*, 32 (1985), 687-693.



## 优先队列（堆）

**虽**然发送到打印机的作业一般都放到队列中，但这未必总是最好的做法。例如，可能有一项作业特别重要，因此希望只要打印机一有空闲就来处理这项作业。反过来，若在打印机有空时正好有多个单页的作业及一项100页的作业等待打印，则更合理的做法也许是最后处理长的作业，尽管它不是最后提交上来的（可是，大多数系统并不这么做，有时可能特别令人烦恼）。

类似地，在多用户环境中，操作系统调度程序必须决定在若干进程中运行哪个进程。一般一个进程只能被允许运行一个固定的时间片。一种算法是使用队列。开始时作业被放到队列的末尾。调度程序将反复提取队列中的第一个作业并运行它，直到运行完毕或者该作业的时间片用完，若作业未被运行完毕就将其放到队列的末尾。这种策略一般并不太合适，因为一些很短的作业由于一味等待运行而要花费很长的时间去处理。一般说来，短的作业要尽可能快地结束，这一点很重要，因此在已经被运行的作业当中这些短作业应该拥有优先权。此外，有些作业虽不短小但很重要，也应该拥有优先权。

这种特殊的应用似乎需要一类特殊的队列，称之为**优先队列**（priority queue）。本章中，我们将讨论：

- 优先队列ADT的高效实现。
- 优先队列的使用。
- 优先队列的高级实现。

我们将看到的这类数据结构属于计算机科学中最雅致的一种。

### 6.1 模型

优先队列是至少允许下列两种操作的数据结构：`insert`（插入），它的工作是显而易见的；`deleteMin`（删除最小项），它的工作是找出、返回和删除优先队列中最小的元素<sup>1</sup>。`insert`操作等价于`enqueue`（入队），而`deleteMin`则是队列操作`dequeue`（出队）在优先队列中的等价操作。

如同大多数数据结构一样，有时可能要添加一些其他的操作，但这些添加的操作属于扩展的操作，而不是图6-1所描述的基本模型的一部分。

除了操作系统外，优先队列还有许多应用。在第7章，我们将看到优先队列如何用于外部排序。在**贪心算法**（greedy algorithm）的实现方面优先队列也很重要，该算法通过反复求出最小元来进行计算；在第9章和第10章我们将看到一些特殊的例子。本章将介绍优先队列在离散事件模

1. C++提供两种版本的 `deleteMin`。一个删除最小项，另一个在删除最小项的同时在通过引用传递的对象中存储所删掉的值。



拟中的应用。

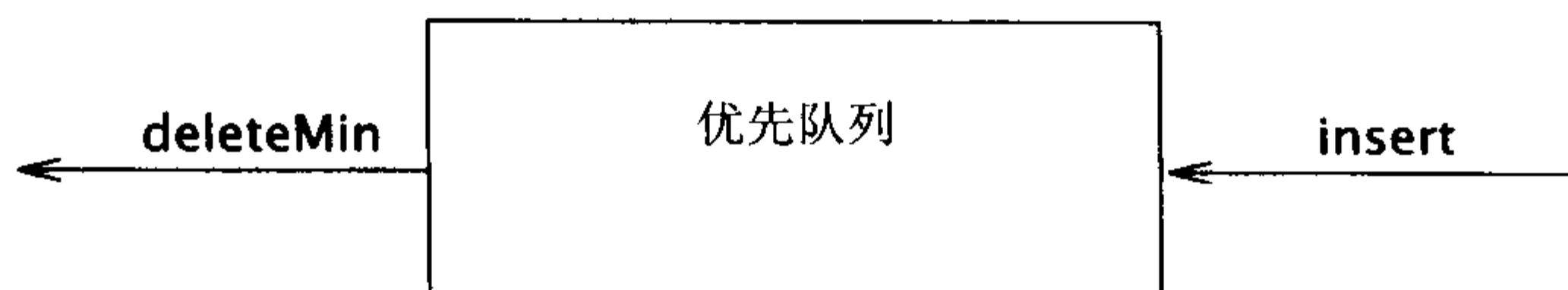


图6-1 优先队列的基本模型

## 6.2 一些简单的实现

有几种明显的方法实现优先队列。我们可以使用一个简单链表在表头以 $O(1)$ 执行插入操作，并遍历该链表以删除最小元，这又需要 $O(N)$ 时间。另一种方法是，始终让表保持排序状态；这使得插入代价高昂（ $O(N)$ ）而`deleteMin`花费低廉（ $O(1)$ ）。基于`deleteMin`的操作从不多于插入操作这一事实，因此前者恐怕是更好的想法。

再一种实现优先队列的方法是使用二叉查找树，它对这两种操作的平均运行时间都是 $O(\log N)$ 。尽管插入是随机的，而删除则不是，但这个结论还是成立的。记住我们删除的唯一元素是最小元。反复除去左子树中的结点似乎损害了树的平衡，使得右子树加重。然而，右子树是随机的。在最坏情形，即`deleteMin`将左子树删空的情形下，右子树拥有的元素最多也就是它应具有的两倍。这只是在其期望的深度上加了一个小常数。注意，通过使用平衡树，可以把界变成最坏情形的界；这将防止出现不好的插入序列。

使用查找树可能有些过分，因为它支持大量并不需要的操作。我们将要使用的基本的数据结构不需要链，它以最坏情形时间 $O(\log N)$ 支持上述两种操作。插入实际上将花费常数平均时间，若无删除干扰，该结构的实现将以线性时间建立一个具有 $N$ 项的优先队列。然后，我们将讨论如何实现优先队列以支持有效的合并。这个附加的操作似乎有些复杂，它显然需要使用链接的结构。

214

## 6.3 二叉堆

我们将要使用的这种工具叫作**二叉堆**（binary heap）。对于优先队列的实现，二叉堆的使用很常见，如果堆（heap）这个词不加修饰地用在优先队列上下文中，一般都是指这种数据结构的实现。在本节，我们把二叉堆只叫作堆。与二叉查找树一样，堆也有两个性质，即结构性性质和堆序性质。类似于AVL树，对堆的操作可能破坏其中一个性质，因此，堆的操作必须到堆的所有性质都被满足时才能终止。事实上这并不难做到。

### 6.3.1 结构性性质

堆是一棵被完全填满的二叉树，可能的例外是在底层，底层上的元素从左到右填入。这样的树称为**完全二叉树**（complete binary tree）。图6-2是一个例子。

容易证明，一棵高为 $h$ 的完全二叉树有 $2^h$ 到 $2^{h+1} - 1$ 个结点。这意味着，完全二叉树的高是 $\lfloor \log N \rfloor$ ，显然它是 $O(\log N)$ 。

一个重要的观察发现，因为完全二叉树很有规律，所以可以用一个数组表示而不需要使用链。图6-3中的数组对应图6-2中的堆。

215



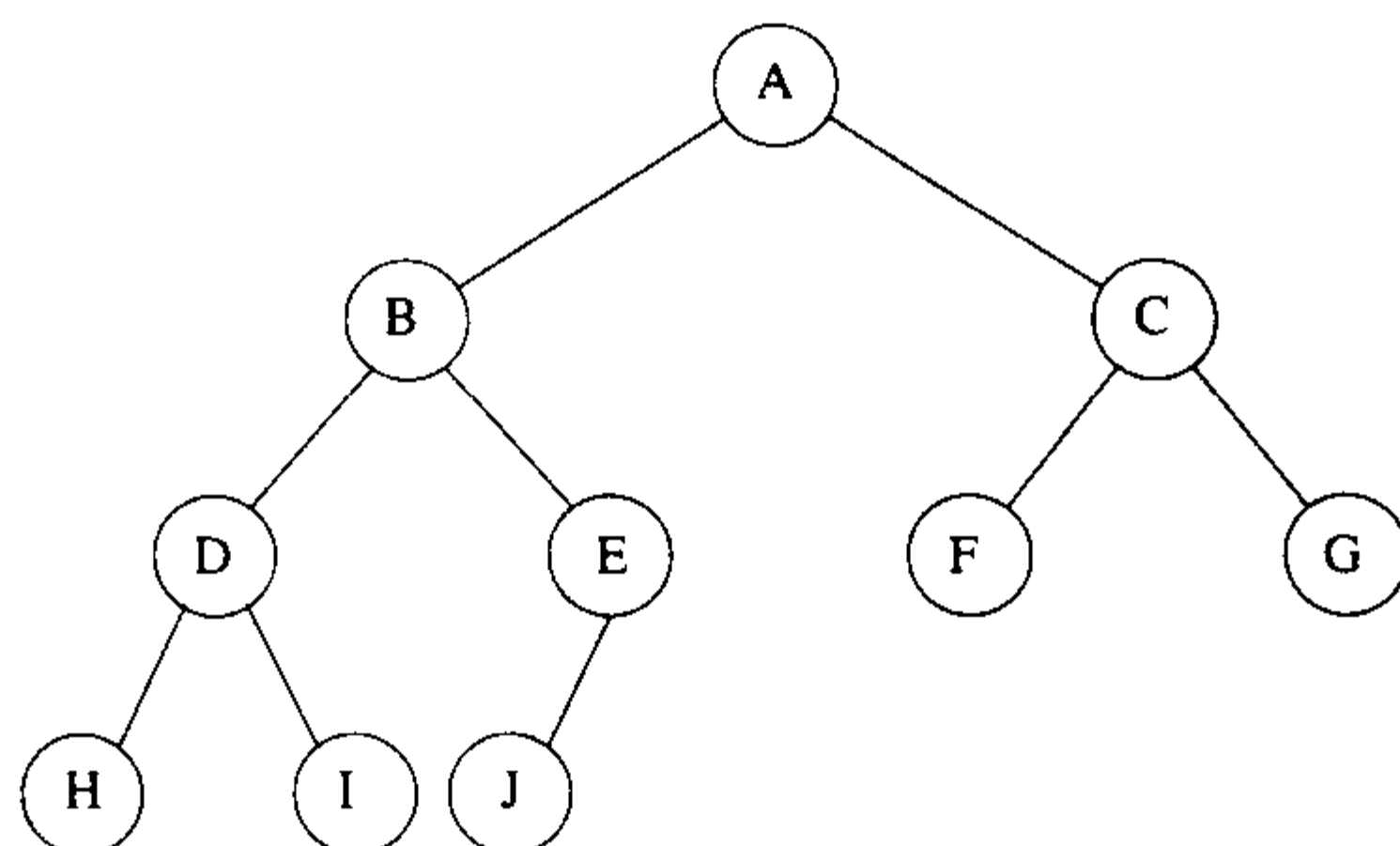


图6-2 一棵完全二叉树

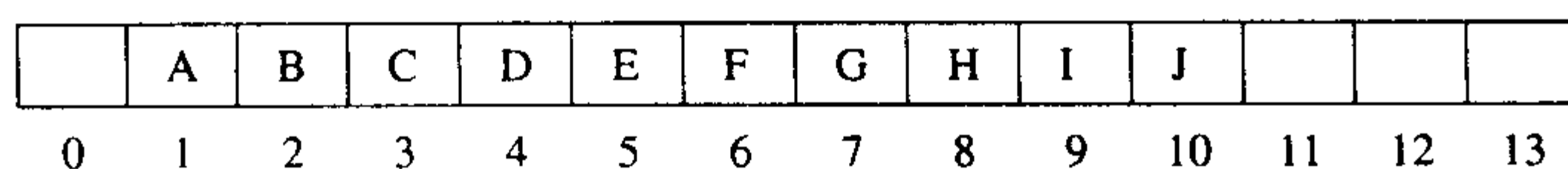


图6-3 完全二叉树的数组实现

对于数组中任一位置 $i$ 上的元素，其左儿子在位置 $2i$ 上，右儿子在左儿子后的单元 $(2i+1)$ 中，它的父亲则在位置 $\lfloor i/2 \rfloor$ 上。因此，这里不仅不需要链，而且遍历该树所需要的操作也极简单，在大部分计算机上都可能运行得非常快。这种实现方法的唯一问题在于，最大的堆大小需要事先估计，但一般情况下这并不成问题（而且如果需要我们可以重新调整）。在图6-3中，堆大小的界限是13个元素。该数组有一个位置0；后面将详细叙述。

因此，一个堆数据结构将由一个（Comparable对象的）数组和一个代表当前堆的大小的整数组成。图6-4所示为一个优先队列接口。

```

1  template <typename Comparable>
2  class BinaryHeap
3  {
4  public:
5      explicit BinaryHeap( int capacity = 100 );
6      explicit BinaryHeap( const vector<Comparable> & items );
7
8      bool isEmpty( ) const;
9      const Comparable & findMin( ) const;
10
11     void insert( const Comparable & x );
12     void deleteMin( );
13     void deleteMin( Comparable & minItem );
14     void makeEmpty( );
15
16 private:
17     int          currentSize; // Number of elements in heap
18     vector<Comparable> array; // The heap array
19
20     void buildHeap( );
21     void percolateDown( int hole );
22 };
  
```

图6-4 优先队列的类接口

本章将始终把堆画成树，这意味着，具体的实现将使用简单的数组。

### 6.3.2 堆序性质

使操作可以快速执行的性质是堆序性质（heap-order property）。由于想要快速地找出最小元，

因此最小元应该在根上。如果考虑任意子树也应该是堆，那么任意结点就应该小于它的所有后裔。

应用这个逻辑，可以得到堆序性质。在堆中，对于每一个结点 $X$ ， $X$ 的父亲中的键小于（或等于） $X$ 中的键，根结点除外（它没有父亲）<sup>1</sup>。在图6-5中左边的树是一个堆，但是，右边的树却不是（虚线表示堆序性质被破坏）。

216

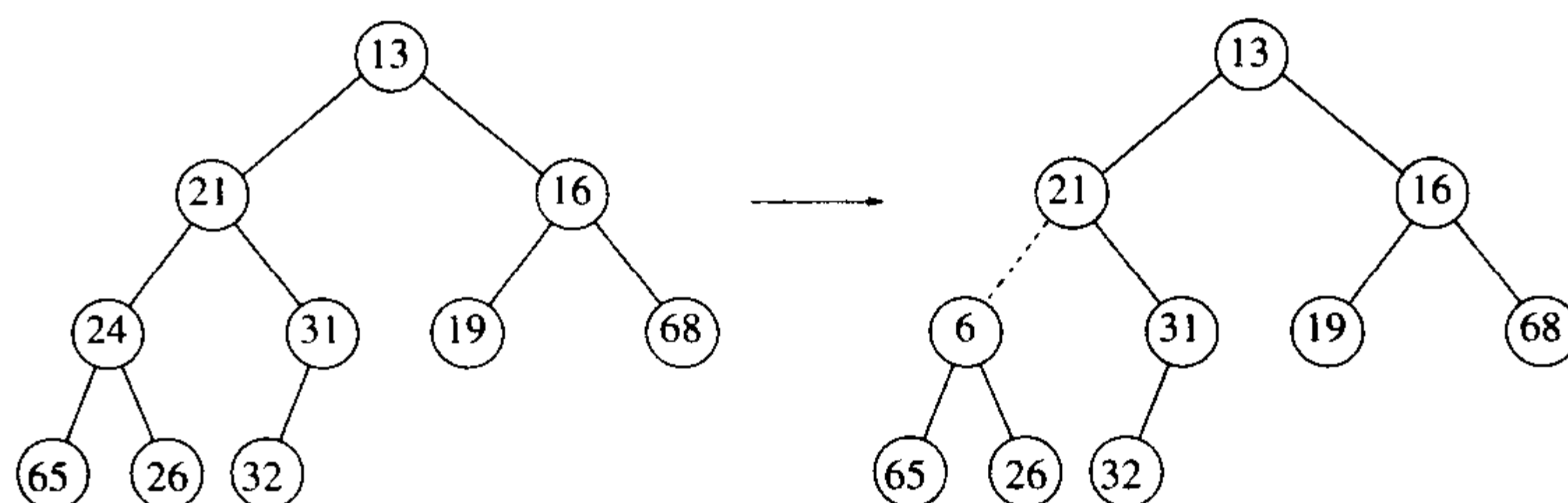


图6-5 两棵完全树（只有左边的树是堆）

根据堆序性质，最小元总可以在根处找到。因此，我们以常数时间得到附加操作findMin。

### 6.3.3 基本的堆操作

无论从概念上还是实际上考虑，执行两个所要求的操作都是很容易的。所有的工作需要保证始终保持堆序性质。

#### 1. insert

为将一个元素 $X$ 插入到堆中，我们在下一个空闲位置创建一个空穴，因为否则该堆将不是完全树。如果 $X$ 可以放在空穴中而并不破坏堆序，那么插入完成。否则，把空穴的父结点上的元素移入空穴中，这样，空穴就朝着根的方向上行一步。继续该过程直到 $X$ 能被放入空穴中为止。图6-6表示，为了插入14，我们在堆的下一个可用位置建立空穴。由于将14插入空穴破坏了堆序性质，因此将31移入空穴。在图6-7中继续这种策略，直到找出置入14的正确位置。

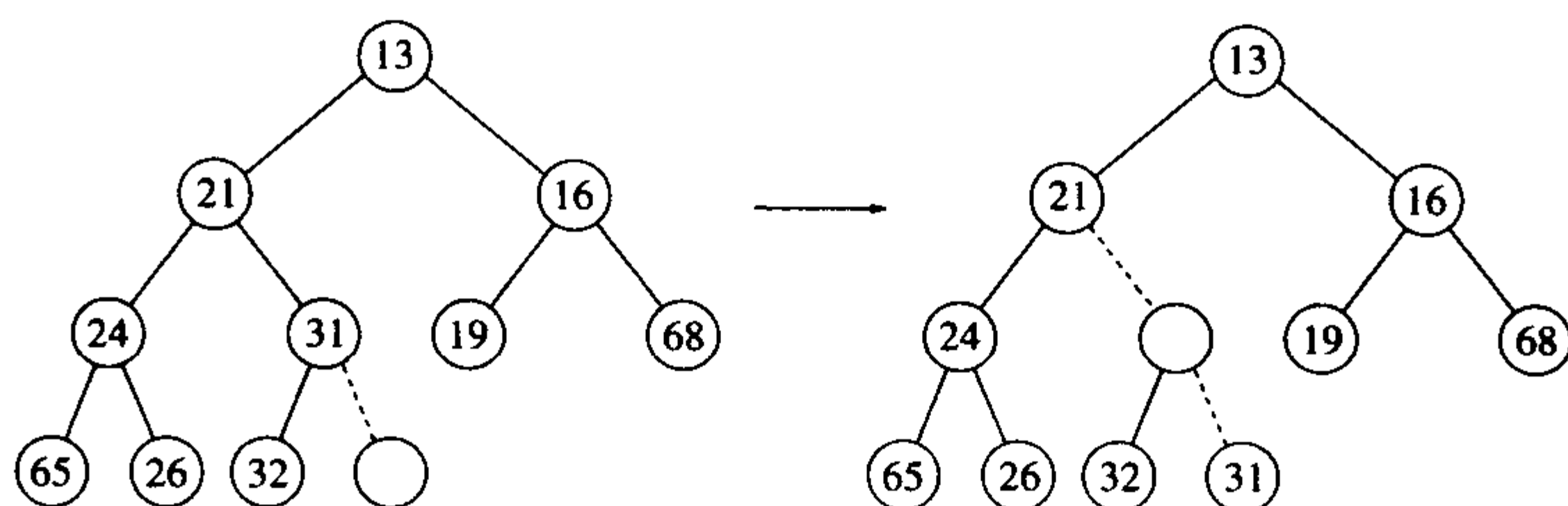


图6-6 尝试插入14：创建一个空穴，再将空穴上冒

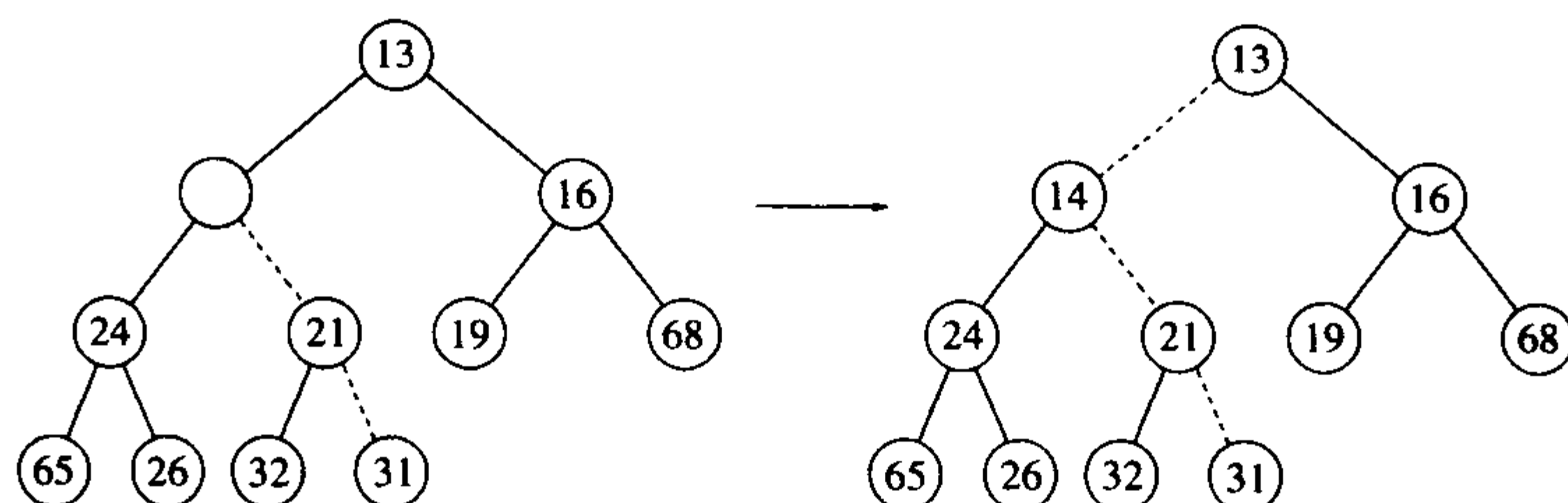


图6-7 将14插入到前面的堆中的最后两步

1. 类似地，我们可以声明一个（*max*）堆，它使我们通过改变堆序性质能够有效地找出和删除最大元。因此，优先队列可以用来找出最大元或最小元，但这需要提前决定。

这种一般的策略叫作上滤（percolate up）：新元素在堆中上滤直到找出正确的位置。使用图6-8所示的代码很容易实现插入。

```

1  /**
2   * Insert item x, allowing duplicates.
3   */
4  void insert( const Comparable & x )
5  {
6      if( currentSize == array.size( ) - 1 )
7          array.resize( array.size( ) * 2 );
8
9      // Percolate up
10     int hole = ++currentSize;
11     for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
12         array[ hole ] = array[ hole / 2 ];
13     array[ hole ] = x;
14 }

```

图6-8 插入到一个二叉堆的过程

其实我们本可以使用insert例程通过反复实施交换操作直至建立正确的顺序来实现上滤过程，可是一次交换需要3条赋值语句。如果一个元素上滤 $d$ 层，那么由于交换而实施的赋值的次数就达到 $3d$ ，而这里的方法却只需要 $d+1$ 次赋值。

217

如果要插入的元素是新的最小值，那么它将一直被推向顶端。这样在某一时刻hole将是1，并且需要程序跳出循环。当然可以用显式测试做到这一点，或者把插入的项复制到位置0，来将程序终止。这里选择显式测试。

如果要插入的元素是新的最小元从而一直上滤到根处，那么这种插入的时间为 $O(\log N)$ 。平均来看，这种上滤终止得要早；业已证明，执行一次插入平均需要2.607次比较，因此insert将元素平均上移1.607层。

## 2. deleteMin

deleteMin以类似于插入的方式处理。找出最小元是容易的；困难的部分是删除它。当删除一个最小元时，要在根结点建立一个空穴。由于现在堆少了一个元素，因此堆中最后一个元素 $X$ 必须移动到该堆的某个地方。如果 $X$ 可以被放到空穴中，那么deleteMin完成。不过这一般不太可能，因此我们将空穴的两个儿子中的较小者移入空穴，这样就把空穴向下推了一层。重复该步骤直到 $X$ 可以被放入空穴中。因此，我们的作法是将 $X$ 置入沿着从根开始包含最小儿子的一条路径上的一个正确位置。

218

图6-9中左边的图显示deleteMin之前的堆。删除13后，必须要正确地将31放到堆中。31不能放在空穴中，因为这将破坏堆序性质。于是，把较小的儿子14置入空穴，同时空穴下滑一层（见图6-10）。重复该过程，由于31大于19，因此把19置入空穴，在更下一层上建立一个新的空穴。然后，再把26置入空穴，在底层又建立一个新的空穴。最后，我们得以将31置入空穴中（见图6-11）。这种一般的策略叫作下滤（percolate down）。在其实现例程中我们使用类似于insert例程中用过的技巧来避免进行交换操作。

在堆的实现中经常发生的错误是，当堆中存在偶数个元素时，将出现一个结点只有一个儿子的情况。我们必须以结点不总有两个儿子为前提，因此这就涉及一个附加的测试。在图6-12描述的程序中，在第40行进行了这种测试。一种极其巧妙的解决方法是始终保证算法把每一个结点都看成有两个儿子。为了实施这种解法，当堆的大小为偶数时，在每个下滤开始处，可将其值大于堆中任何元素的标记放到堆的终端后面的位置上。必须在深思熟虑以后再这么做，而且必须判断

219

是否确实需要使用这种技巧。虽然这不再需要测试右儿子是否存在，但还是需要测试何时到达底层，因为对每一片树叶，算法将需要一个标记。

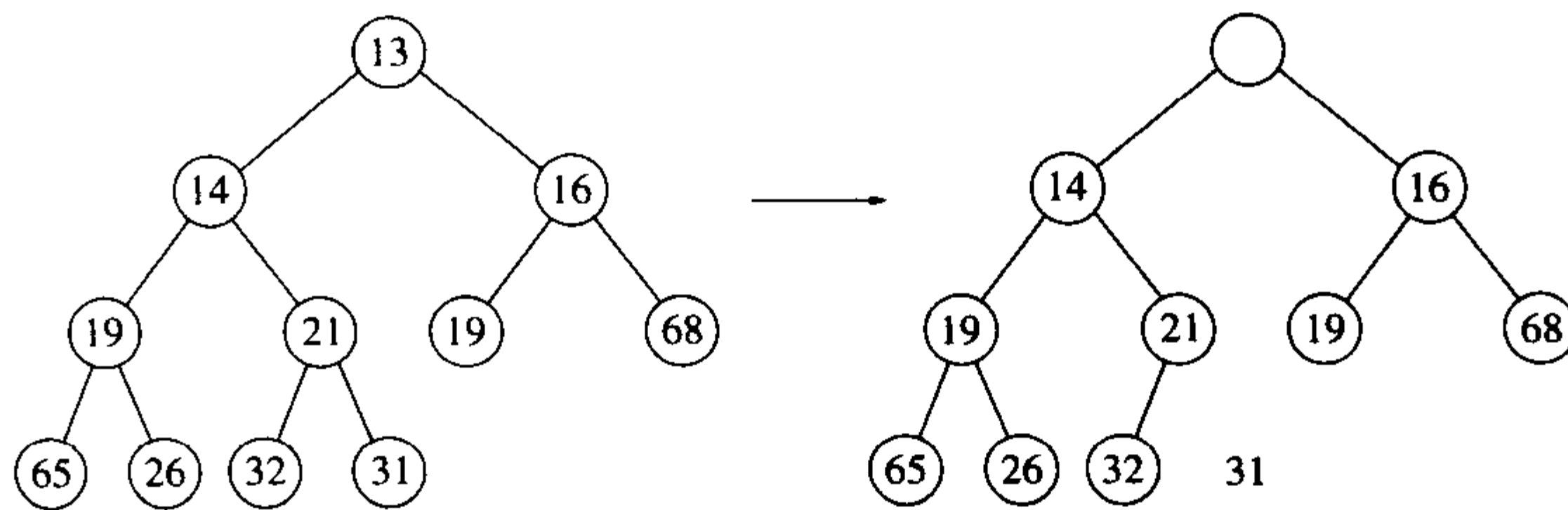


图6-9 在根处建立空穴

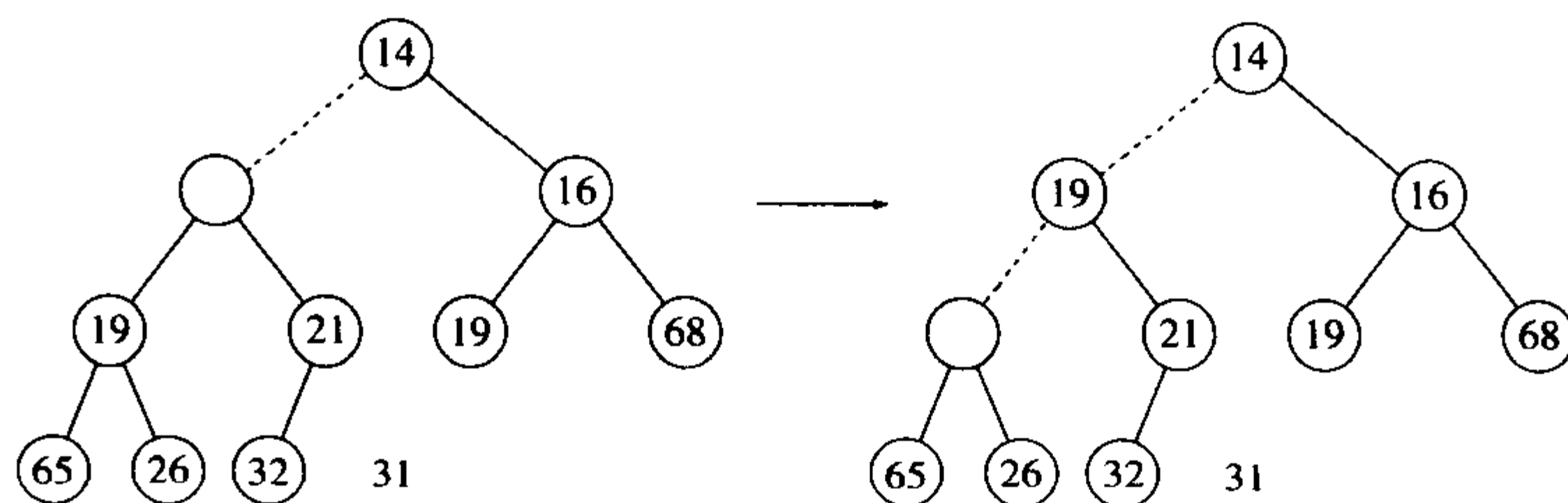


图6-10 在deleteMin中的接下来的两步

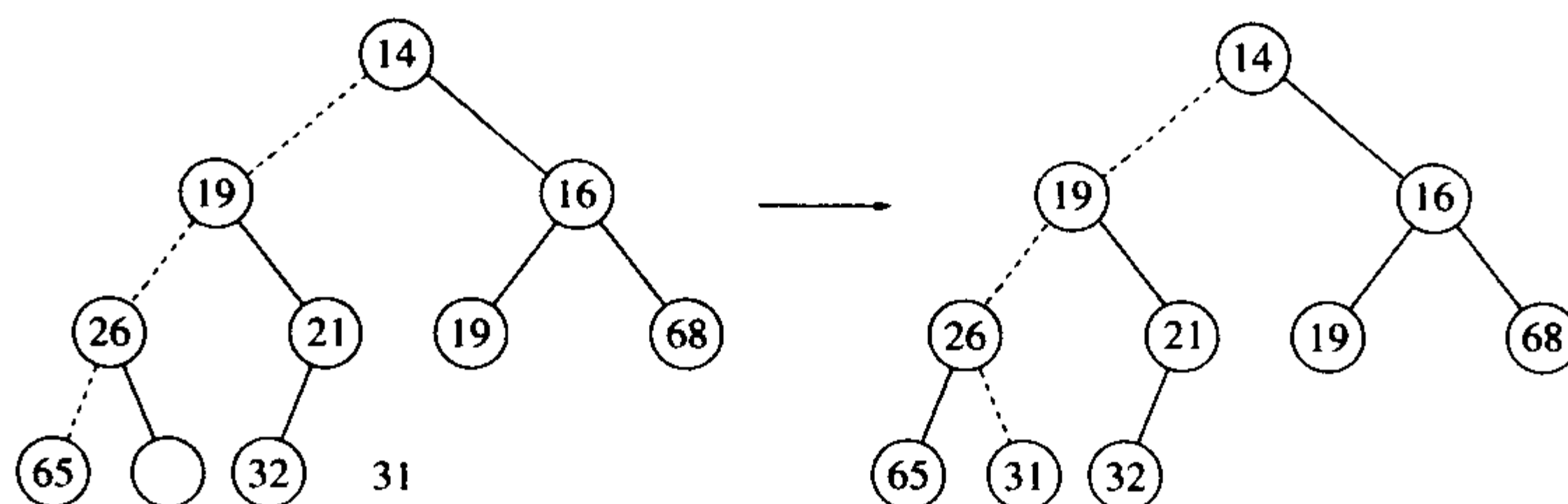


图6-11 在deleteMin中的最后两步

```

1  /**
2   * Remove the minimum item.
3   * Throws UnderflowException if empty.
4   */
5  void deleteMin( )
6  {
7      if( isEmpty( ) )
8          throw UnderflowException( );
9
10     array[ 1 ] = array[ currentSize-- ];
11     percolateDown( 1 );
12 }
13
14 /**
15 * Remove the minimum item and place it in minItem.
16 * Throws UnderflowException if empty.
17 */
18 void deleteMin( Comparable & minItem )
19 {
20     if( isEmpty( ) )
21         throw UnderflowException( );

```

图6-12 在二叉堆中执行deleteMin的方法



```

22
23     minItem = array[ 1 ];
24     array[ 1 ] = array[ currentSize-- ];
25     percolateDown( 1 );
26 }
27
28 /**
29  * Internal method to percolate down in the heap.
30  * hole is the index at which the percolate begins.
31  */
32 void percolateDown( int hole )
33 {
34     int child;
35     Comparable tmp = array[ hole ];
36
37     for( ; hole * 2 <= currentSize; hole = child )
38     {
39         child = hole * 2;
40         if( child != currentSize && array[ child + 1 ] < array[ child ] )
41             child++;
42         if( array[ child ] < tmp )
43             array[ hole ] = array[ child ];
44         else
45             break;
46     }
47     array[ hole ] = tmp;
48 }

```

图6-12 在二叉堆中执行deleteMin的方法（续）

这种操作的最坏情形运行时间为 $O(\log N)$ 。平均而言，被放到根处的元素几乎下滤到堆的底层（即它所来自的那层），因此平均运行时间为 $O(\log N)$ 。

### 6.3.4 堆的其他操作

注意，虽然求最小值操作可以在常数时间完成，但是，按照求最小元设计的堆（也称做最小堆（min heap））在求最大元方面却无任何作用。事实上，一个堆所蕴涵的关于序的信息很少，因此，若不对整个堆进行线性搜索，是没有办法找出任何特定的元素的。为说明这一点，观察图6-13所示的大型堆结构（具体元素没有标出），可以看到，关于最大元所知道的唯一信息是：该元素在其中一片树叶上。但是，半数的元素都在树叶上，因此该信息是没有多大用途的。由于这个原因，如果了解元素的位置信息很重要的话，那么除堆之外，还必须用到诸如散列表等某些其他的数据结构（回忆一下：该模型并不允许查看堆内部）。

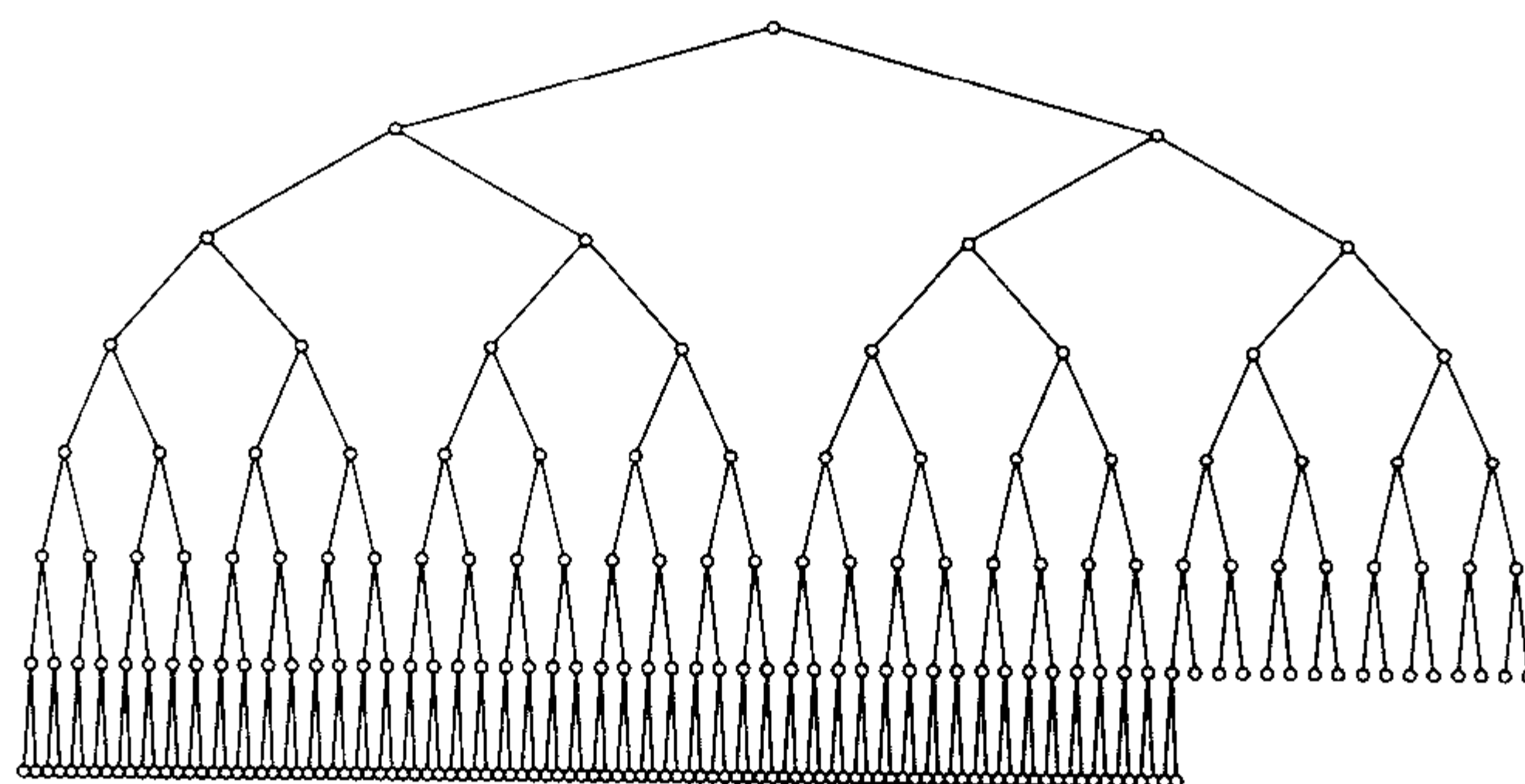


图6-13 一棵巨大的完全二叉树

如果假设通过某种其他方法得知每一个元素的位置，那么其他几种操作的开销就很小。下述的前三种操作均以对数最坏情形时间运行。

### 1. decreaseKey

decreaseKey( $p, \Delta$ )操作减小在位置 $p$ 处的元素的值，减小的幅度为正的量 $\Delta$ 。由于这可能破坏堆序性质，因此必须通过上滤操作对堆进行调整。该操作对系统管理程序是有用的：系统管理程序能够使它们的程序以最高的优先级来运行。

### 2. increaseKey

increaseKey( $p, \Delta$ )操作增加在位置 $p$ 处的元素的值，增加的幅度为正的量 $\Delta$ 。这可以用下滤来完成。许多调度程序自动地降低过多消耗CPU时间的进程的优先级。

### 3. remove

remove( $p$ )操作删除堆中位置 $p$ 上的结点。这通过首先执行decreaseKey( $p, \infty$ )然后再执行deleteMin()来完成。当一个进程由用户中止（而不是正常终止）时，必须将其从优先队列中除去。

### 4. buildHeap

有时候二叉堆通过项的原始集合来构造。这个构造函数将 $N$ 项作为输入并把它们放入一个堆中。很明显，这可以通过 $N$ 次连续的insert来完成。由于每个insert操作都花费 $O(1)$ 的平均时间，以及 $O(\log N)$ 的最坏情形时间，这个算法的总的运行时间就是 $O(N)$ 平均时间，其最坏情形时间为 $O(N \log N)$ 。由于这是一种特殊的指令而且没有其他的操作介入，并且已知该指令可以在线性平均时间内完成，那么合理的线性时间界是可以保证的。

通常的算法是将 $N$ 项以任意顺序放入树中，并保持结构特性。此时，如果percolateDown( $i$ )从结点 $i$ 下滤，那么图6-14中的buildHeap例程则生成一棵堆序的树（heap-ordered tree）。

```

1 explicit BinaryHeap( const vector<Comparable> & items )
2   : array( items.size( ) + 10 ), currentSize( items.size( ) )
3 {
4     for( int i = 0; i < items.size( ); i++ )
5         array[ i + 1 ] = items[ i ];
6     buildHeap( );
7 }
8
9 /**
10  * Establish heap order property from an arbitrary
11  * arrangement of items. Runs in linear time.
12  */
13 void buildHeap( )
14 {
15     for( int i = currentSize / 2; i > 0; i-- )
16         percolateDown( i );
17 }

```

图6-14 buildHeap和构造函数

图6-15中的第一棵树是无序树。从图6-15到图6-18中其余7棵树表示出7个percolateDown中每一个的执行结果。每条虚线对应两次比较：一次是找出较小的儿子结点，另一次是将较小的儿子与该结点比较。注意，在整个算法中只有10条虚线（可能已经存在第11条——在哪里？），它们对应20次比较。

为了确定buildHeap的运行时间的界，我们必须确定虚线的条数的界。这可以通过计算堆中所有结点的高度和来得到，它是虚线的最大条数。现在我们想要说明的是：该和为 $O(N)$ 。

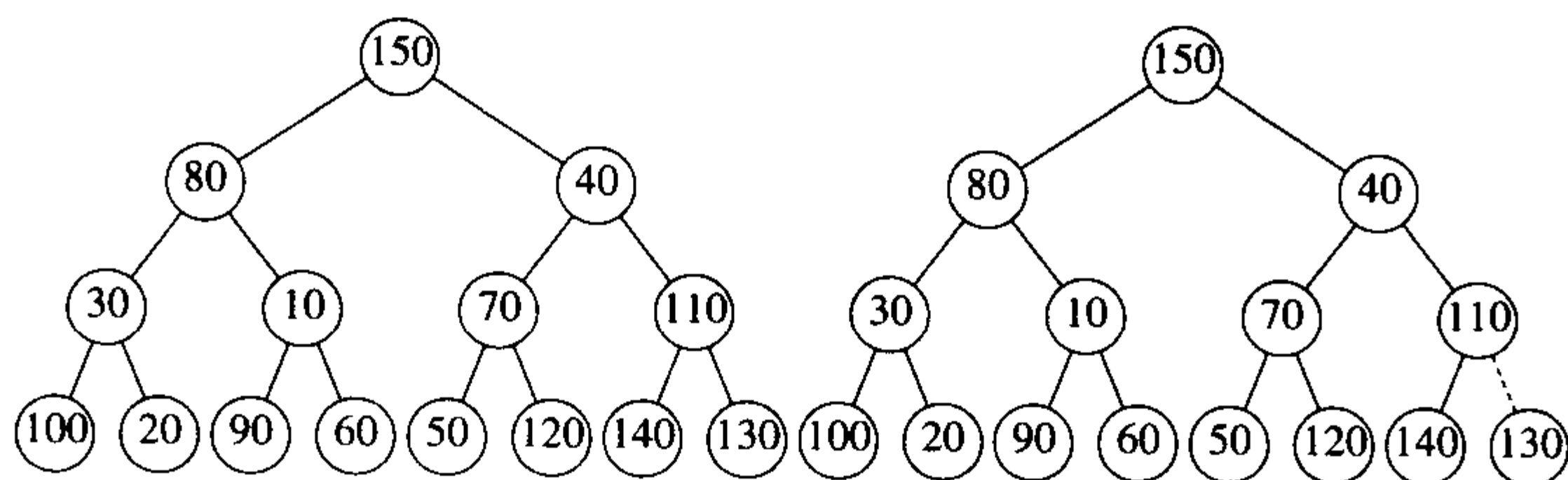


图6-15 左——初始堆; 右——在percolateDown(7)之后

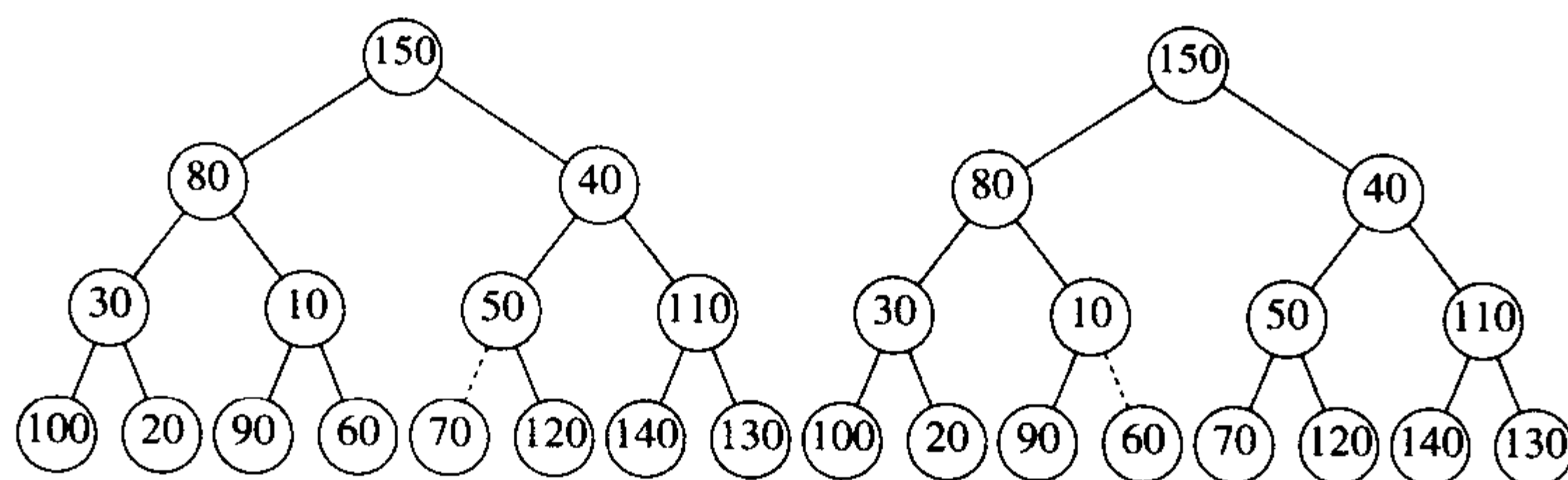


图6-16 左——percolateDown(6)之后; 右——percolateDown(5)之后

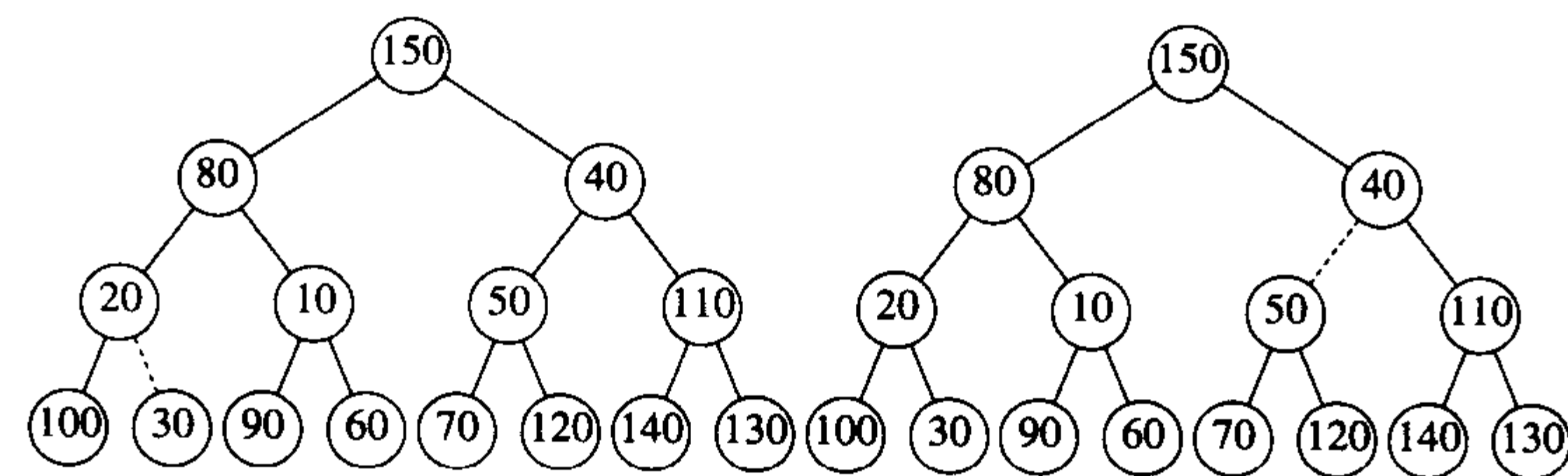


图6-17 左——percolateDown(4)之后; 右——percolateDown(3)之后

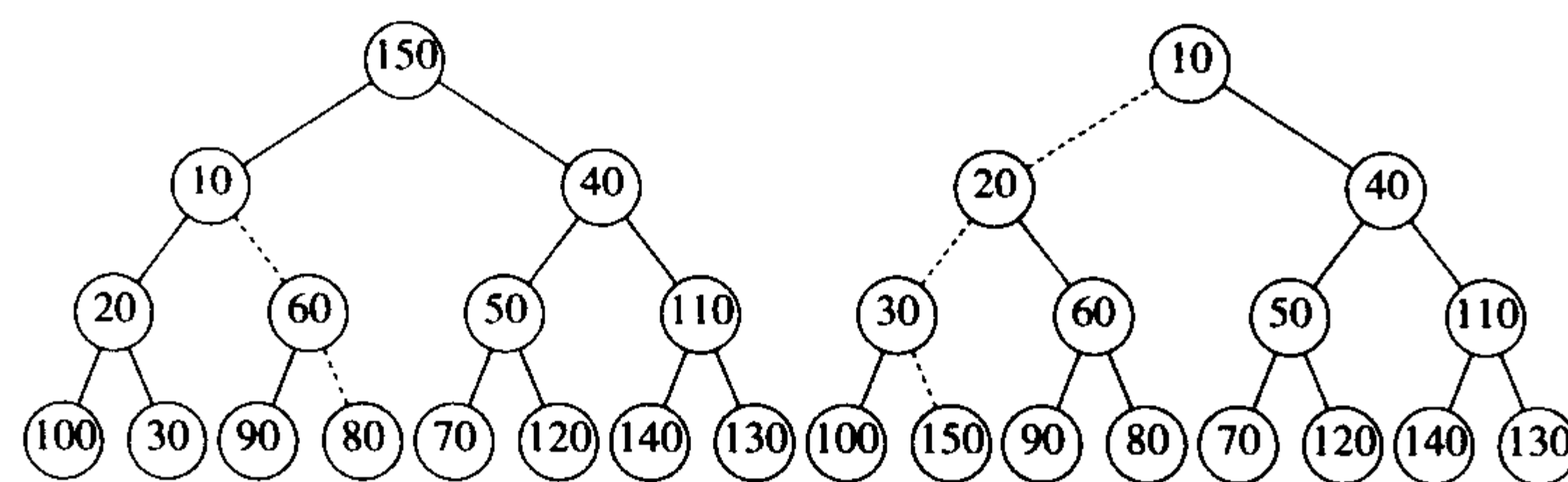


图6-18 左——percolateDown(2)之后; 右——percolateDown(1)之后

**定理6.1** 包含 $2^{h+1} - 1$ 个结点且高为 $h$ 的理想二叉树(perfect binary tree)的结点的高度和为 $2^{h+1} - 1 - (h + 1)$ 。

**证明** 容易看出, 该树由高度 $h$ 上的1个结点、高度 $h-1$ 上的2个结点、高度 $h-2$ 上的 $2^2$ 个结点以及一般在高度 $h-i$ 上的 $2^i$ 个结点组成。则所有结点的高度和为

$$S = \sum_{i=0}^h 2^i (h-i)$$

$$= h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \cdots + 2^{h-1}(1) \quad (6.1)$$

两边乘以2得到方程

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \cdots + 2^h(1) \quad (6.2)$$

将这两个方程相减得到方程(6.3)。我们发现, 非常数项几乎都消去了, 例如我们有 $2h - 2(h-1) = 2$ ,

$4(h-1) - 4(h-2) = 4$ , 等等。方程 (6.2) 的最后一项  $2^h$  在方程 (6.1) 中不出现; 因此, 它出现在方程 (6.3) 中。方程 (6.1) 中的第一项  $h$  在方程 (6.2) 中不出现; 因此,  $-h$  出现在方程 (6.3) 中。我们得到

$$S = -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h + 1) \quad (6.3)$$

定理得证。 ■

完全树 (complete tree) 不是理想二叉树 (perfect binary tree), 但是我们得到的结果却是完全树的结点高度和的上界。由于完全树的结点数在  $2^h$  和  $2^{h+1}$  之间, 因此该定理意味着这个和是  $O(N)$ , 其中  $N$  是结点的个数。

虽然我们得到的结果对证明 buildHeap 是线性的而言是充分的, 但是高度和的界却不是尽可能强。对于具有  $N = 2^h$  个结点的完全树, 我们得到的界大致是  $2N$ 。由归纳法可以证明, 高度和是  $N - b(N)$ , 其中  $b(N)$  是在  $N$  的二进制表示法中 1 的个数。

## 6.4 优先队列的应用

我们已经提到优先队列如何用于操作系统的设计中。在第9章, 我们将看到优先队列如何有效地用于几个图论算法的实现中。下面将介绍如何应用优先队列来得到两个问题的解答。

225

### 6.4.1 选择问题

我们将要考察的第一个问题是来自第1章的选择问题 (selection problem)。回忆当时的输入是  $N$  个元素以及一个整数  $k$ , 这  $N$  个元素的集可以是全序集。该选择问题是要找出第  $k$  个最大的元素。

在第1章中给出了两个算法, 但都不是很有有效的算法。第一个算法我们将称其为 1A, 将这些元素读入数组并将它们排序, 返回需要的元素。假设使用的是简单的排序算法, 则运行时间为  $O(N^2)$ 。另一个算法这里称为 1B, 是将  $k$  个元素读入一个数组并将其排序。这些元素中的最小者在第  $k$  个位置上。我们依次处理其余的元素。处理一个元素时, 先将该元素与数组中第  $k$  个元素进行比较, 如果该元素大, 那么将第  $k$  个元素删除, 而将这个新元素放在其余  $k-1$  个元素间的正确的位置上。当算法结束时, 第  $k$  个位置上的元素就是正确答案。该方法的运行时间为  $O(N \cdot k)$  (为什么? )。如果  $k = \lceil N/2 \rceil$ , 那么这两种算法都是  $O(N^2)$ 。注意, 对于任意的  $k$ , 我们可以求解对称的问题: 找出第  $(N-k+1)$  个最小的元素, 从而  $k = \lceil N/2 \rceil$  实际上是这两个算法要处理的最苛刻的情况。这恰好也是最有趣的情形, 因为此时的这个  $k$  值称为中位数 (median)。

我们在这里给出两个算法, 在  $k = \lceil N/2 \rceil$  的极端情形下, 这两个算法都以  $O(M \log N)$  时间运行, 这是明显的改进。

#### 1. 算法 6A

为了简单起见, 假设我们只考虑找出第  $k$  个最小元素。该算法很简单。先将  $N$  个元素读入一个数组, 然后对该数组应用 buildHeap 算法。最后, 执行  $k$  次 deleteMin 操作。最后从该堆提取的元素就是正确答案。显然, 只要改变堆序性质, 就可以求解原始的问题: 找出第  $k$  个最大的元素。

这个算法的正确性应该是很明显的。如果使用 buildHeap, 构造堆的最坏情形是  $O(N)$  时间, 而每次执行 deleteMin 是  $O(\log N)$  时间。由于有  $k$  次 deleteMin, 因此我们得到总的运行时间为



$O(N + k \log N)$ 。如果  $k = O(N/\log N)$ ，那么运行时间取决于 `buildHeap` 操作，即  $O(N)$ 。对于大的  $k$  值，运行时间为  $O(k \log N)$ 。如果  $k = \lceil N/2 \rceil$ ，那么运行时间则为  $\Theta(M \log N)$ 。

注意，令  $k = N$ ，运行该程序并在元素离开堆时记录下这些元素的值，那么实际上就是对输入文件以时间  $O(M \log N)$  进行了排序。在第7章，我们将细化该想法，并得到一种快速的排序算法，称为堆排序（heapsort）。

## 2. 算法6B

226

对于第2个算法，我们回到原始问题，找出第  $k$  个最大的元素。使用算法1B的思路。在任一时刻都将维持  $k$  个最大元素的集合  $S$ 。在前  $k$  个元素读入以后，当再读入一个新的元素时，该元素将与第  $k$  个最大元素进行比较，设第  $k$  个最大的元素为  $S_k$ 。注意， $S_k$  是  $S$  中最小的元素。如果新的元素更大，那么用新元素代替  $S$  中的  $S_k$ 。此时， $S$  将有一个新的最小元素，它可能是新添加进来的元素，也可能不是。在输入完成时，我们找到  $S$  中的最小元素，将其返回，它就是该问题的答案。

这基本上与第1章中描述的算法相同。不过，这里我们使用一个堆来实现  $S$ 。通过调用 `buildHeap` 将前  $k$  个元素以总时间  $O(k)$  放入堆中。处理其余的每个元素的时间为  $O(1)$ ，还要加上  $O(\log k)$  时间，用来检测元素是否放入  $S$ ，并在需要时删除  $S_k$  并插入新元素。因此，总的运行时间是  $O(k + (N - k) \log k) = O(N \log k)$ 。该算法也给出找出中位数的时间界  $\Theta(M \log N)$ 。

在第7章，我们将探讨如何以平均时间  $O(N)$  解决这个问题。在第10章，则讨论一个以  $O(N)$  最坏情形时间  $O(N)$  求解该问题的算法，虽然这个算法有点不切实际但却很雅致。

## 6.4.2 事件模拟

3.7.3节描述了一个重要的排队问题。设有一个系统，比如银行，顾客们到达并排队等候，直到在总计  $k$  个出纳员中有一个出纳员有空。顾客的到达情况由概率分布函数控制，服务时间（当出纳员有空时，用于服务的时间量）也是如此。我们关心的是平均一位顾客要等多久或所排的队伍可能有多长这类统计问题。

对于某些概率分布以及  $k$  的一些值，答案可以精确地计算出来。然而随着  $k$  的变大，显然分析变得越来越困难，因此使用计算机模拟银行的运作就很必要。用这种方法，银行官员可以确定为保证合理顺畅的服务需要多少出纳员。

模拟由处理中的事件组成。这里的两个事件是：(a) 一位顾客的到达；(b) 一位顾客的离去，从而腾出一名出纳员。

我们可以使用概率函数来生成一个输入流，它由每位顾客的到达时间和服务时间的序偶组成，并以到达时间排序。我们不必使用一天中的准确时间，而是使用一份单位时间量，称之为一个滴答（tick）。

进行这种模拟的一个方法是在0滴答处启动一台模拟钟表。让钟表一次走一个滴答，同时查看是否有事件发生。如果有，那么处理这个（或这些）事件，搜集统计资料。当没有顾客留在输入流中且所有的出纳员都闲着的时候，模拟结束。

这种模拟策略的问题是，其运行时间不依赖于顾客数或事件数（每位顾客有两个事件），却依赖于滴答数，而后者实际并不是输入的一部分。下面来看为什么这很重要：假设将钟表的单位改成毫滴答（millitick）并将输入中的所有时间乘以1000，则结果将是：模拟用时长1000倍！

避免这种问题的关键是在每一个阶段让钟表直接走到下一个事件时间。从概念上看这是容易做到的。在任一时刻，可能出现的下一事件是：(a) 输入文件中的下一位顾客到达；(b) 在一名出纳员处一位顾客离开。由于可以得知事件发生的所有时间，因此只需找出最近要发生的事件并

处理这个事件即可。

如果事件是离开，那么处理过程包括搜集离开的顾客的统计资料以及检验队伍（队列），看是否还有其他顾客在等待。如果有，那么加上这位顾客，处理所需要的统计资料，计算顾客将要离开的时间，并将离开事件加到等待发生的事件集中去。

如果事件是到达，那么检查是否有空闲的出纳员。若没有，就把这一到达事件放到队伍（队列）中去；否则，分配给顾客一个出纳员，计算顾客的离开时间，并将离开事件加到等待发生的事件集中去。

在等待的顾客队伍可以实现为一个队列。由于需要找到最近将要发生的事件，因此合适的办法是将等待发生的离开事件的集合编入一个优先队列中。下一事件是下一个到达或下一个离开（哪个先发生那么就是下一个事件）；它们都容易做到。

虽然可能很耗时，但编写这样一个模拟例程还是很简单的。如果有  $C$  个顾客（因此有  $2C$  个事件）和  $k$  个出纳员，那么模拟的运行时间将是  $O(C \log(k+1))$ ，因为计算和处理每个事件花费  $O(\log H)$ ，其中  $H = k+1$ ，为堆的大小<sup>1</sup>。

## 6.5 $d$ 堆

二叉堆是如此简单，以至于在需要优先队列的时候几乎总是使用它。 $d$  堆是二叉堆的简单推广，它与二叉堆很像，但其所有的结点都有  $d$  个儿子（因此，二叉堆是 2 堆）。

图 6-19 所示为一个 3 堆。注意， $d$  堆要比二叉堆浅得多，它将 insert 操作的运行时间改进为  $O(\log_d N)$ 。然而，对于大的  $d$  值，deleteMin 操作就费时得多。因为虽然树浅了，但是必须要找出  $d$  个儿子中最小的一个。如果使用标准算法，则需要进行  $d-1$  次比较，于是将操作的运行时间提高到  $O(d \log_d N)$ 。如果  $d$  是常数，那么当然两种情况的运行时间都是  $O(\log N)$ 。虽然仍然可以使用一个数组，但是，现在找出儿子和父亲的乘法和除法都有个因子  $d$ ，除非  $d$  是 2 的幂，否则将会由于不能通过二进制移位来实现除法而导致运行时间的急剧增加。 $d$  堆在理论上很有趣，因为存在许多算法，其插入次数比 deleteMin 的次数多很多（因此理论上的加速是可能的）。当优先队列太大不能完全装入主存的时候， $d$  堆也是很有用的。在这种情况下， $d$  堆可以与 B 树大致相同的方式发挥作用。最后，有证据显示，在实践中 4 堆可以胜过二叉堆。

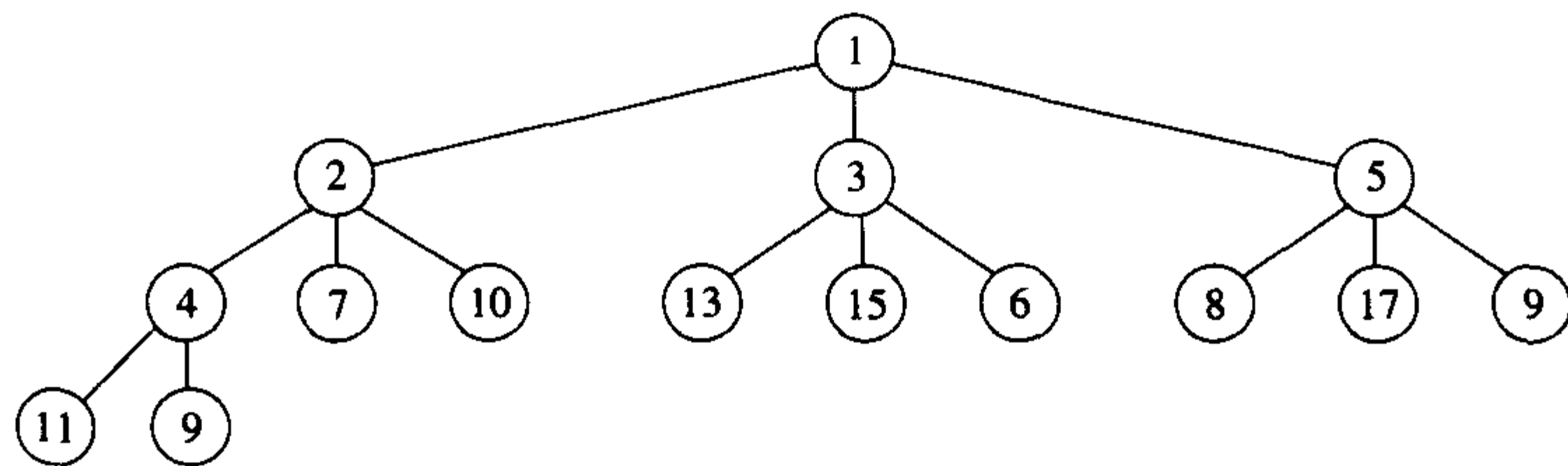


图 6-19 一个  $d$  堆

除了不能执行 find 操作外，堆的实现的最明显的缺点是：将两个堆合而为一是很困难的操作。这个附加的操作称为合并（merge）。有许多实现堆的方法可以使一次 merge 操作的运行时间是  $O(\log N)$ 。下面讨论三种复杂程度不一的数据结构，它们都有效地支持 merge 操作。我们将把复杂的分析推迟到第 11 章讨论。

1. 我们使用  $O(C \log(k+1))$  代替  $O(C \log k)$ ，以避免  $k=1$  时的混淆。

## 6.6 左式堆

设计一种像二叉堆那样有效地支持合并操作（即以 $o(N)$ 时间处理一个merge）而且只使用一个数组的堆结构似乎很困难。原因在于，合并似乎需要把一个数组拷贝到另一个数组中去，对于相同大小的堆这将花费时间 $\Theta(N)$ 。正因为如此，所有支持有效合并的高级数据结构都需要使用链式数据结构。实践中，预计这可能使得所有其他的操作变慢。

**左式堆**（leftist heap）像二叉堆那样既有结构性质，又有堆序性质。事实上，和所有使用的堆一样，左式堆具有相同的堆序性质，该性质我们已经看到过。不仅如此，左式堆也是二叉树。左式堆和二叉堆唯一的区别是：左式堆不是理想平衡的（perfectly balanced），而且事实上是趋于非常不平衡的。

### 6.6.1 左式堆性质

我们把任一结点 $X$ 的零路径长（null path length） $npl(X)$ 定义为从 $X$ 到一个不具有两个儿子的结点的最短路径的长。因此，具有0个或1个儿子的结点的 $npl$ 为0，而 $npl(\text{NULL}) = -1$ 。在图6-20的树中，零路径长标记在树的结点内。

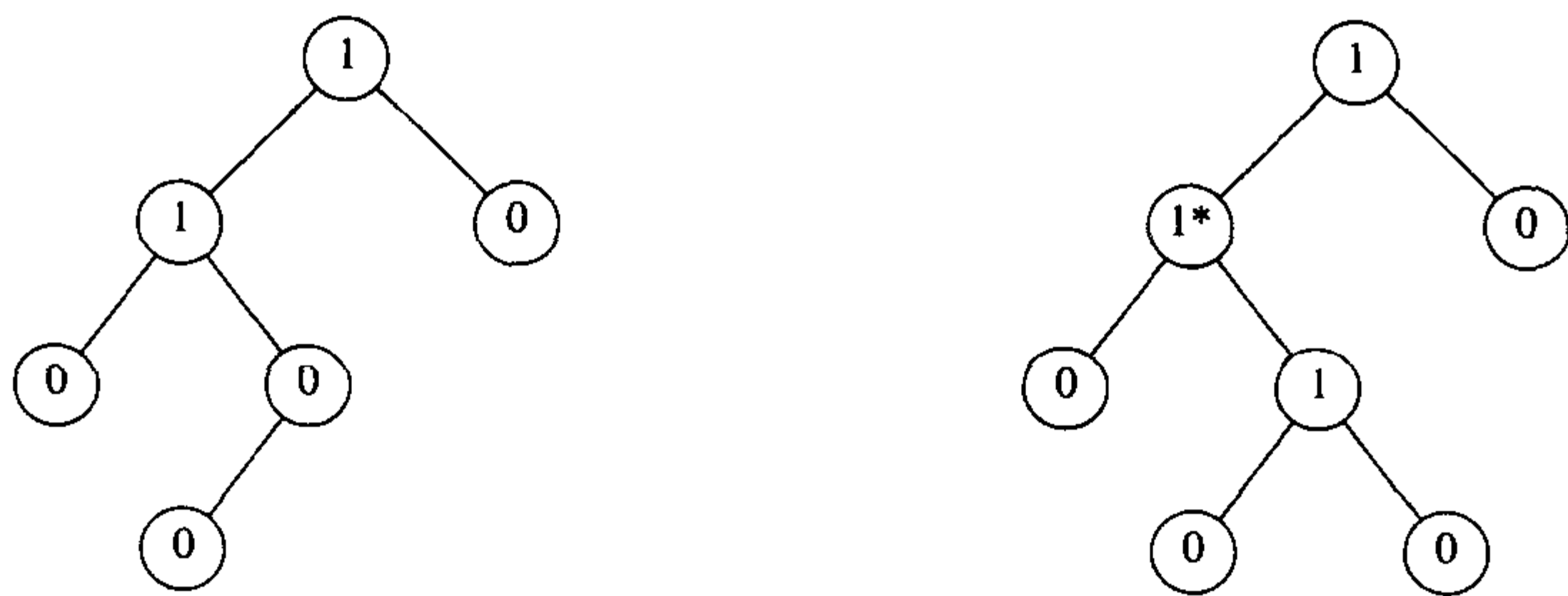


图6-20 两棵树的零路径长；只有左边的树是左式的

注意，任一结点的零路径长比它的诸儿子结点的零路径长的最小值多1。这个结论也适用少于两个儿子的结点，因为null的零路径长是-1。

左式堆性质是：对于堆中的每一个结点 $X$ ，左儿子的零路径长至少与右儿子的零路径长一样大。图6-20中只有一棵树（即左边的那棵树）满足该性质。这个性质实际上超出了它确保树不平衡的要求，因为它显然偏重于使树向左增加深度。确实有可能存在由左结点形成的长路径构成的树（而且实际上更便于合并操作）——因此，称之为**左式堆**（leftist heap）。

因为左式堆趋向于加深左路径，所以右路径应该短。事实上，沿左式堆右侧的右路径确实是该堆中最短的路径。否则，就会存在一条路径通过某个结点 $X$ 并取得左儿子，此时 $X$ 就破坏了左式堆的性质。

229

**定理6.2** 在右路径上有 $r$ 个结点的左式树必然至少有 $2^r - 1$ 个结点。

**证明** 数学归纳法证明。如果 $r = 1$ ，则必然至少存在一个树结点。其次，设定理对 $1, 2, \dots, r$ 个结点成立。考虑在右路径上有 $r + 1$ 个结点的左式树。此时，根具有在右路径上含 $r$ 个结点的右子树，以及在右路径上至少含 $r$ 个结点的左子树（否则它就不是左式树）。对这两条子树应用归纳假设，得知在每棵子树上最少有 $2^r - 1$ 个结点，再加上根结点，于是在该树上至少有 $2^{r+1} - 1$ 个结点，定理得证。 ■



从这个定理可以得到,  $N$ 个结点的左式树有一条右路径最多含有 $\lfloor \log(N+1) \rfloor$ 个结点。对左式堆操作的一般思路是, 将所有的工作放到右路径上进行, 它保证树深短。唯一的棘手部分在于, 对右路径的insert和merge可能会破坏左式堆性质。事实上, 恢复该性质是非常容易的。

## 6.6.2 左式堆操作

对左式堆的基本操作是合并。注意, 插入只是合并的特殊情形, 因为可以把插入看成是单结点堆与一个大的堆的merge。首先, 我们给出一个简单的递归解法, 然后介绍如何非递归地施行该解法。我们的输入是两个左式堆 $H_1$ 和 $H_2$ , 见图6-21。可以验证, 这些堆确实是左式堆。注意, 最小的元素在根处。除数据、左指针和右指针所用空间外, 每个单元还要有一个指示零路径长的项。

230

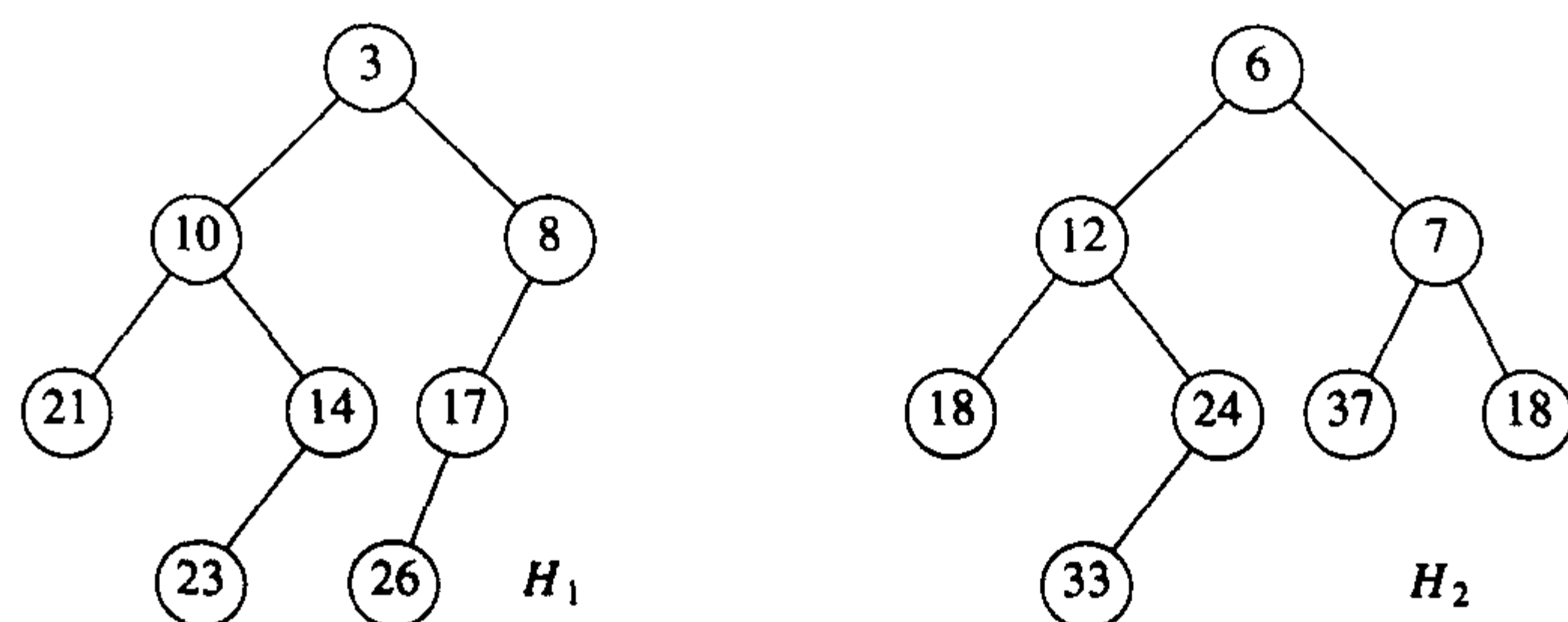


图6-21 两个左式堆 $H_1$ 和 $H_2$

如果这两个堆中有一个堆是空的, 那么可以返回另外一个堆。否则, 为合并这两个堆, 需要比较它们的根。首先, 递归地将具有大的根值的堆与具有小的根值的堆的右子堆合并。在本例中, 递归地将 $H_2$ 与 $H_1$ 中根在8处的右子堆合并, 得到图6-22中的堆。

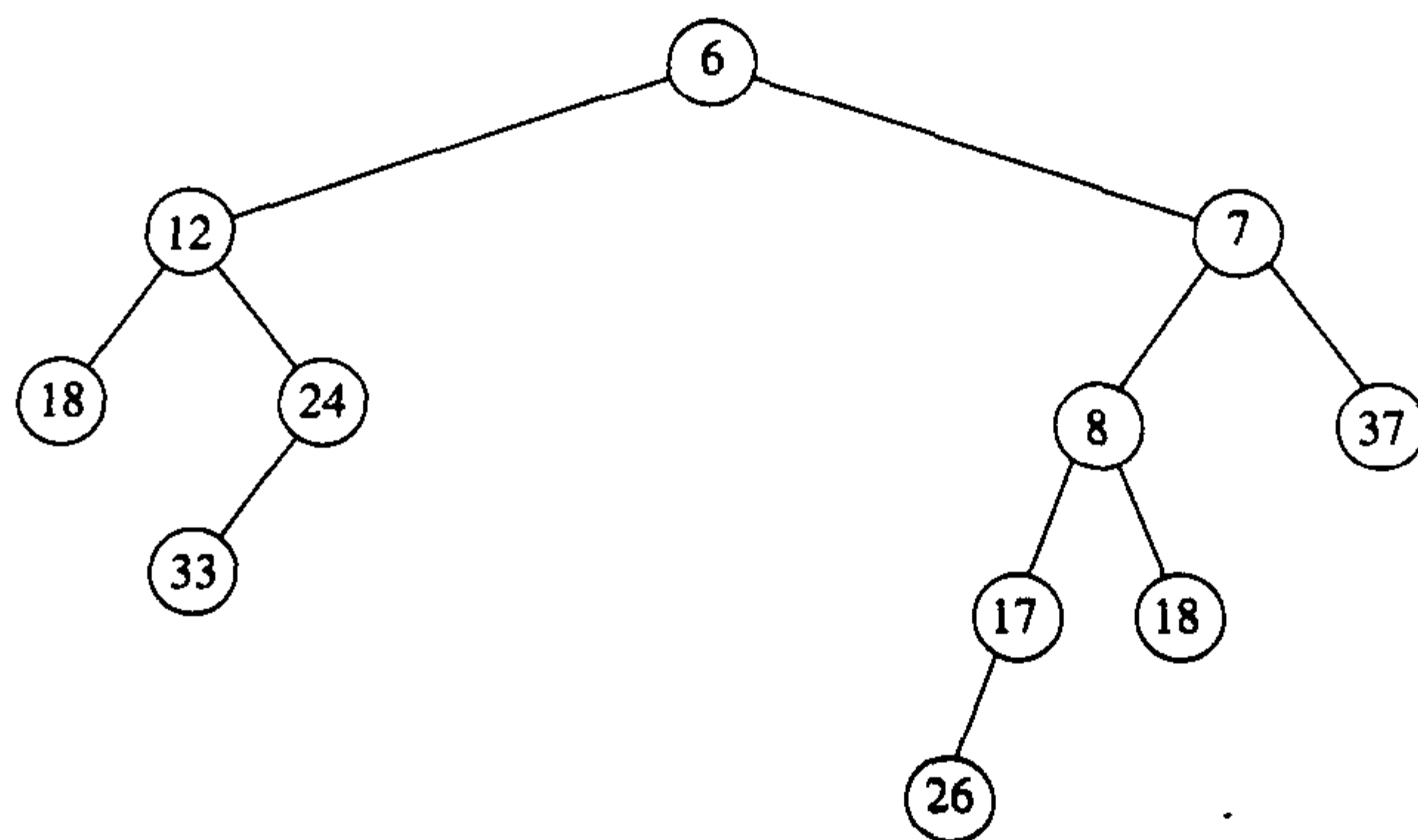
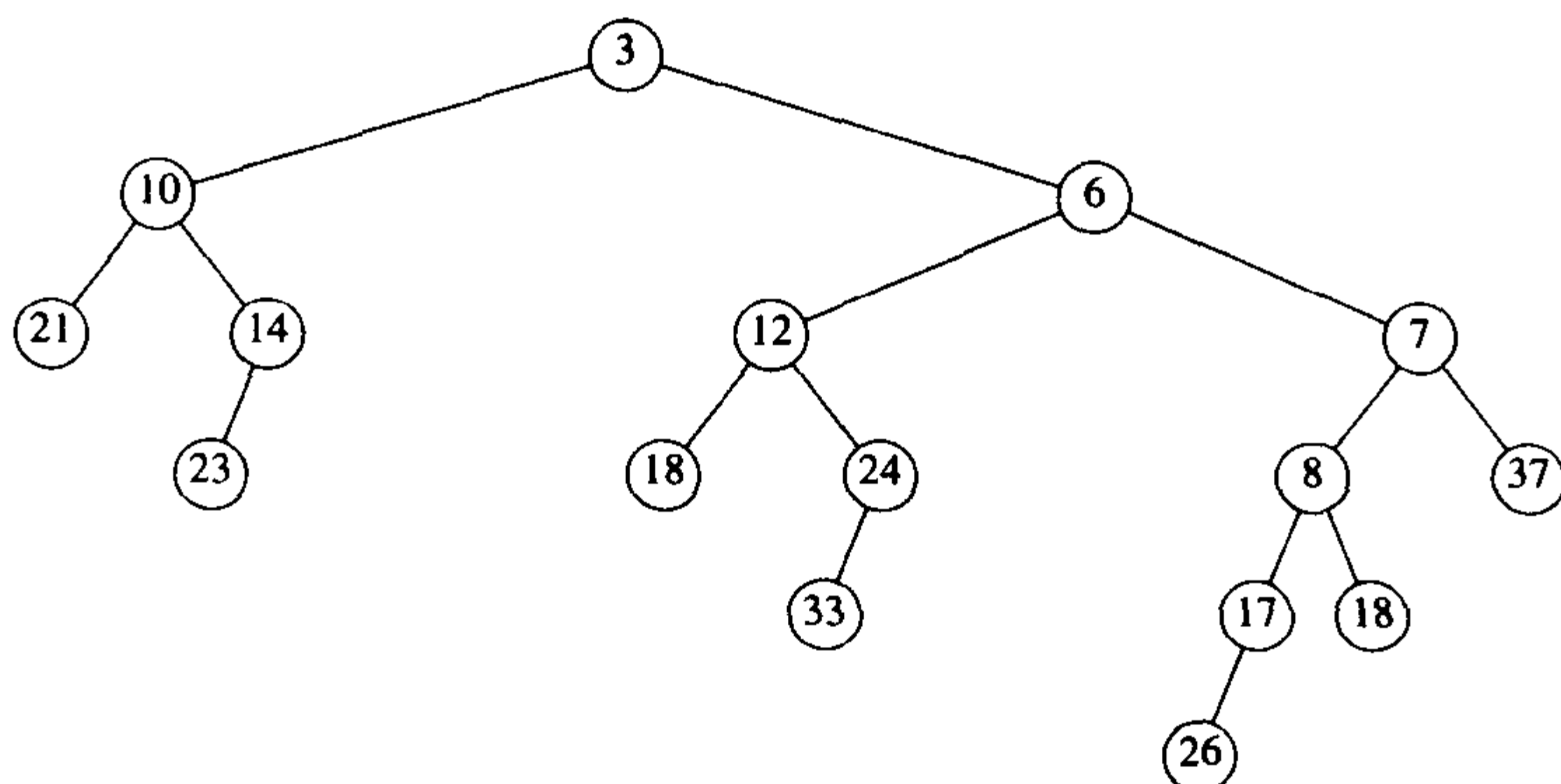


图6-22 将 $H_2$ 与 $H_1$ 的右子堆合并的结果

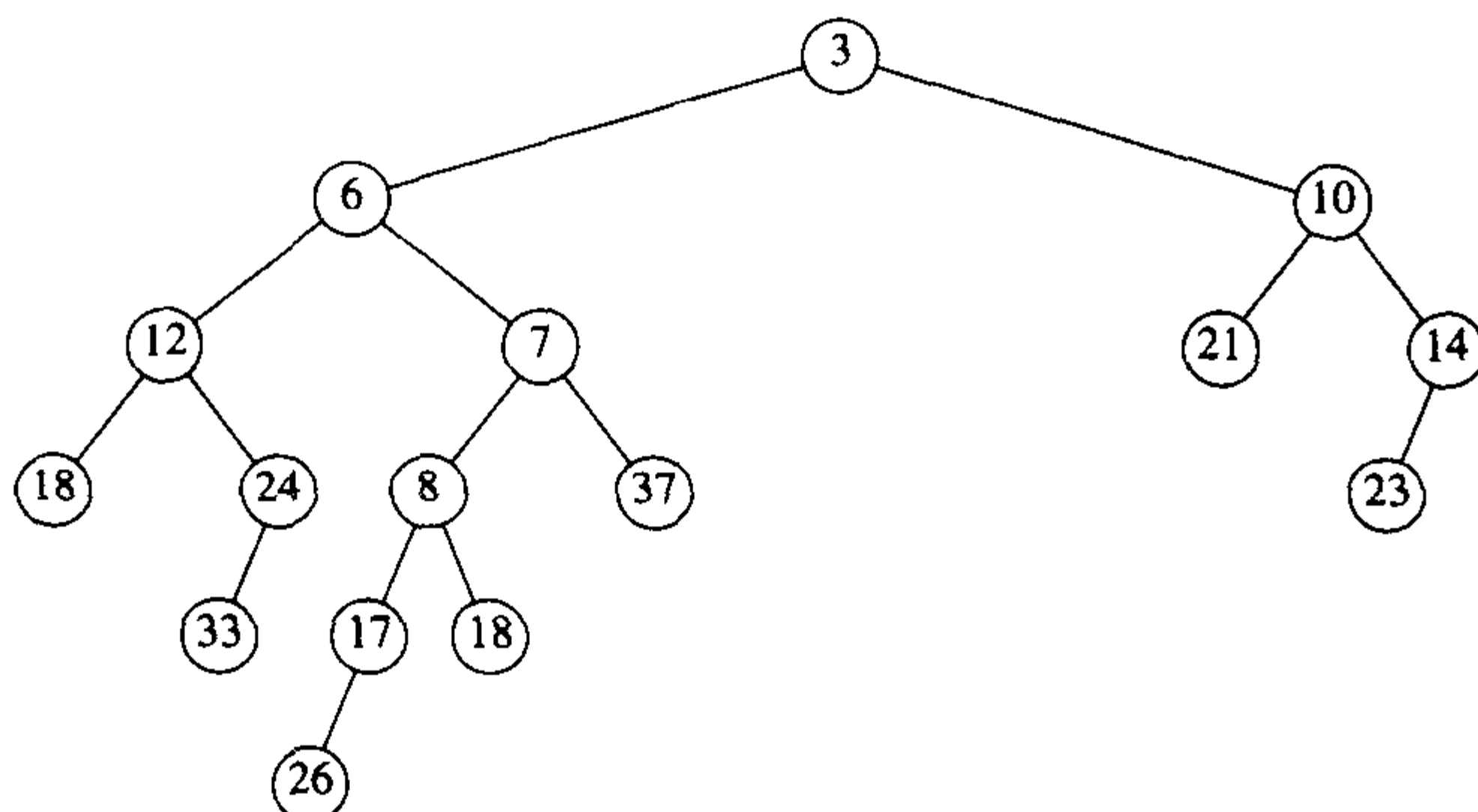
这棵树是递归地形成的, 但我们尚未对算法描述完毕, 因此, 现在还不能说明该堆是如何得到的。不过, 可以假设, 最后的结果是一个左式堆, 因为它通过递归的步骤得到的。这很像归纳法证明中的归纳假设。既然我们能够处理基准情形 (发生在一棵树是空的时候), 当然可以假设, 只要我们能够完成合并那么递归步骤就是成立的; 这是在第1章中讨论过的递归法则3。现在, 我们让这个新的堆成为 $H_1$ 的根的右儿子 (见图6-23)。

231



图6-23 将前面图中的左式堆作为 $H_1$ 的右儿子接上后的结果

虽然最后得到的堆满足堆序性质，但是，它不是左式堆，因为根的左子树的零路径长为1而根的右子树的零路径长为2。因此，左式堆的性质在根处被破坏。不过，容易看到，树的其余部分必然是左式的。由于递归步骤，根的右子树是左式的。根的左子树没有变化，当然它也必然还是左式的。这样一来，只要对根进行调整就可以了。使整棵树是左式的做法如下：只要交换根的左儿子和右儿子（见图6-24）并更新零路径长，就完成了merge，新的零路径长是新的右儿子的零路径长加1。注意，如果零路径长不更新，那么所有的零路径长都将是0，而堆将不是左式的，只是随机的。在这种情况下，算法仍然成立，但是，我们即将声称的时间界将不再有效。

图6-24 交换 $H_1$ 的根的儿子得到的结果

将算法的描述直接翻译成代码。除了增加npl（零路径长）字段外，结点类（见图6-25）与二叉树是相同的。左式堆把对根的指针作为其数据成员存储。我们在第4章已经看到，当一个元素插入到一棵空的二叉树时，需要改变由根引用的结点。我们使用通常的实现private递归方法的技巧进行合并。该类的构架也在图6-25中表示。

```

1 template <typename Comparable>
2 class LeftistHeap
3 {
4     public:
5         LeftistHeap( );
6         LeftistHeap( const LeftistHeap & rhs );
7         ~LeftistHeap( );
8 
```

图6-25 左式堆类型声明

```

9    bool isEmpty( ) const;
10   const Comparable & findMin( ) const;
11
12   void insert( const Comparable & x );
13   void deleteMin( );
14   void deleteMin( Comparable & minItem );
15   void makeEmpty( );
16   void merge( LeftistHeap & rhs );
17
18   const LeftistHeap & operator=( const LeftistHeap & rhs );
19
20 private:
21   struct LeftistNode
22   {
23       Comparable element;
24       LeftistNode *left;
25       LeftistNode *right;
26       int npl;
27
28       LeftistNode( const Comparable & theElement, LeftistNode *lt = NULL,
29                   LeftistNode *rt = NULL, int np = 0 )
30           : element( theElement ), left( lt ), right( rt ), npl( np ) { }
31   };
32
33   LeftistNode *root;
34
35   LeftistNode * merge( LeftistNode *h1, LeftistNode *h2 );
36   LeftistNode * merge1( LeftistNode *h1, LeftistNode *h2 );
37
38   void swapChildren( LeftistNode *t );
39   void reclaimMemory( LeftistNode *t );
40   LeftistNode * clone( LeftistNode *t ) const;
41 };

```

图6-25 左式堆类型声明 (续)

两个merge例程 (见图6-26) 设计成消除一些特殊情形并保证 $H_1$ 有较小根的驱动程序。实际的合并操作在merge1中进行 (见图6-27)。公有的merge方法将rhs合并到控制堆中, rhs变成了空的。这个公有方法中的别名测试不接受h.merge(h)。

232

```

1  /**
2   * Merge rhs into the priority queue.
3   * rhs becomes empty. rhs must be different from this.
4   */
5  void merge( LeftistHeap & rhs )
6  {
7      if( this == &rhs )    // Avoid aliasing problems
8          return;
9
10     root = merge( root, rhs.root );
11     rhs.root = NULL;
12 }
13
14 /**
15  * Internal method to merge two roots.
16  * Deals with deviant cases and calls recursive merge1.
17  */
18 LeftistNode * merge( LeftistNode *h1, LeftistNode *h2 )

```

图6-26 合并左式堆的驱动例程

```

19 {
20     if( h1 == NULL )
21         return h2;
22     if( h2 == NULL )
23         return h1;
24     if( h1->element < h2->element )
25         return merge1( h1, h2 );
26     else
27         return merge1( h2, h1 );
28 }

```

图6-26 合并左式堆的驱动例程(续)

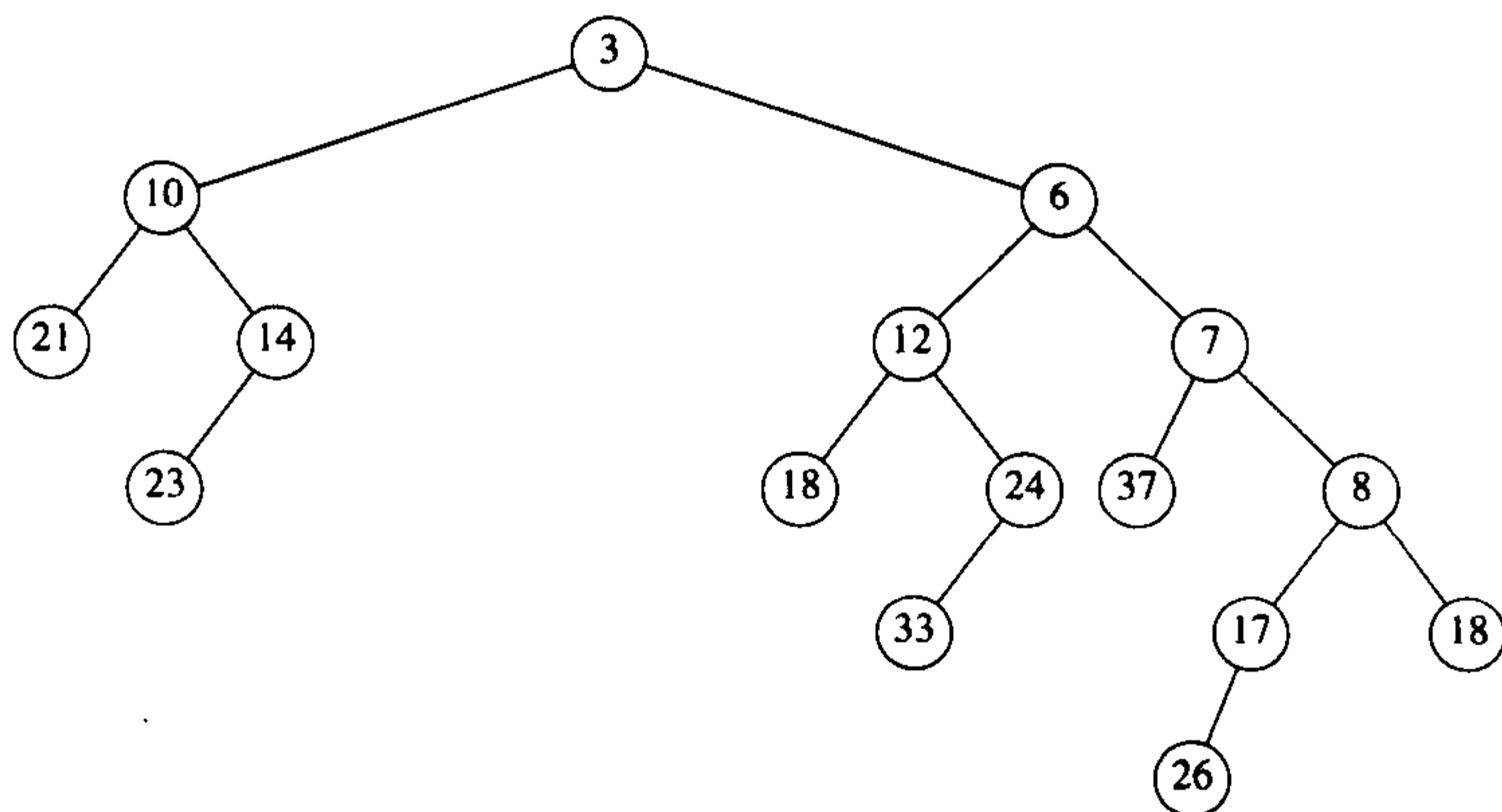
```

1 /**
2  * Internal method to merge two roots.
3  * Assumes trees are not empty, and h1's root contains smallest item.
4  */
5 LeftistNode * merge1( LeftistNode *h1, LeftistNode *h2 )
6 {
7     if( h1->left == NULL ) // Single node
8         h1->left = h2;      // Other fields in h1 already accurate
9     else
10    {
11        h1->right = merge( h1->right, h2 );
12        if( h1->left->npl < h1->right->npl )
13            swapChildren( h1 );
14        h1->npl = h1->right->npl + 1;
15    }
16    return h1;
17 }

```

图6-27 合并左式堆的实际例程

执行合并的时间与右路径的长的和成正比,因为在递归调用期间对每一个被访问的结点执行的是常数工作量。因此,合并两个左式堆的时间界为 $O(\log N)$ 。也可以分两次来非递归地实施该操作。在第一次,通过合并两个堆的右路径建立一棵新的树。为此,以排序的方式安排 $H_1$ 和 $H_2$ 右路径上的结点,保持它们各自的左儿子不变。在我们的例子中,新的右路径是3, 6, 7, 8, 18, 而最后得到的树如图6-28所示。第二次构成堆,儿子的交换工作在左式堆性质被破坏的结点上进行。在图6-28中,在结点7和3各有一次交换,并得到与前面相同的树。非递归的做法更容易理解,但是编程困难。我们留给读者去证明:递归过程和非递归过程的结果是相同的。

图6-28 合并 $H_1$ 和 $H_2$ 的右路径的结果

上面提到，可以通过把插入项看成单结点堆并执行一次merge来完成插入。为了执行deleteMin，只要除掉根而得到两个堆，然后再将这两个堆合并即可。因此，执行一次deleteMin的时间为 $O(\log N)$ 。这两个例程如图6-29和图6-30所示。

```

1 /**
2  * Inserts x; duplicates allowed.
3  */
4 void insert( const Comparable & x )
5 {
6     root = merge( new LeftistNode( x ), root );
7 }

```

图6-29 左式堆的插入例程

```

1 /**
2  * Remove the minimum item.
3  * Throws UnderflowException if empty.
4  */
5 void deleteMin( )
6 {
7     if( isEmpty( ) )
8         throw UnderflowException( );
9
10    LeftistNode *oldRoot = root;
11    root = merge( root->left, root->right );
12    delete oldRoot;
13 }
14
15 /**
16  * Remove the minimum item and place it in minItem.
17  * Throws UnderflowException if empty.
18  */
19 void deleteMin( Comparable & minItem )
20 {
21     minItem = findMin( );
22     deleteMin( );
23 }

```

图6-30 左式堆的deleteMin例程

最后，可以通过建立二叉堆（显然使用链接实现）来以 $O(N)$ 时间建立左式堆。尽管二叉堆显然是左式的，但它未必是最佳解决方案，因为我们得到的堆可能是最差的左式堆。不仅如此，以相反的层序遍历树也不像使用链那么容易。buildHeap的效果可以通过递归地建立左右子树然后将根下滤而达到。练习中包括另外一个解决方案。

233

## 6.7 斜堆

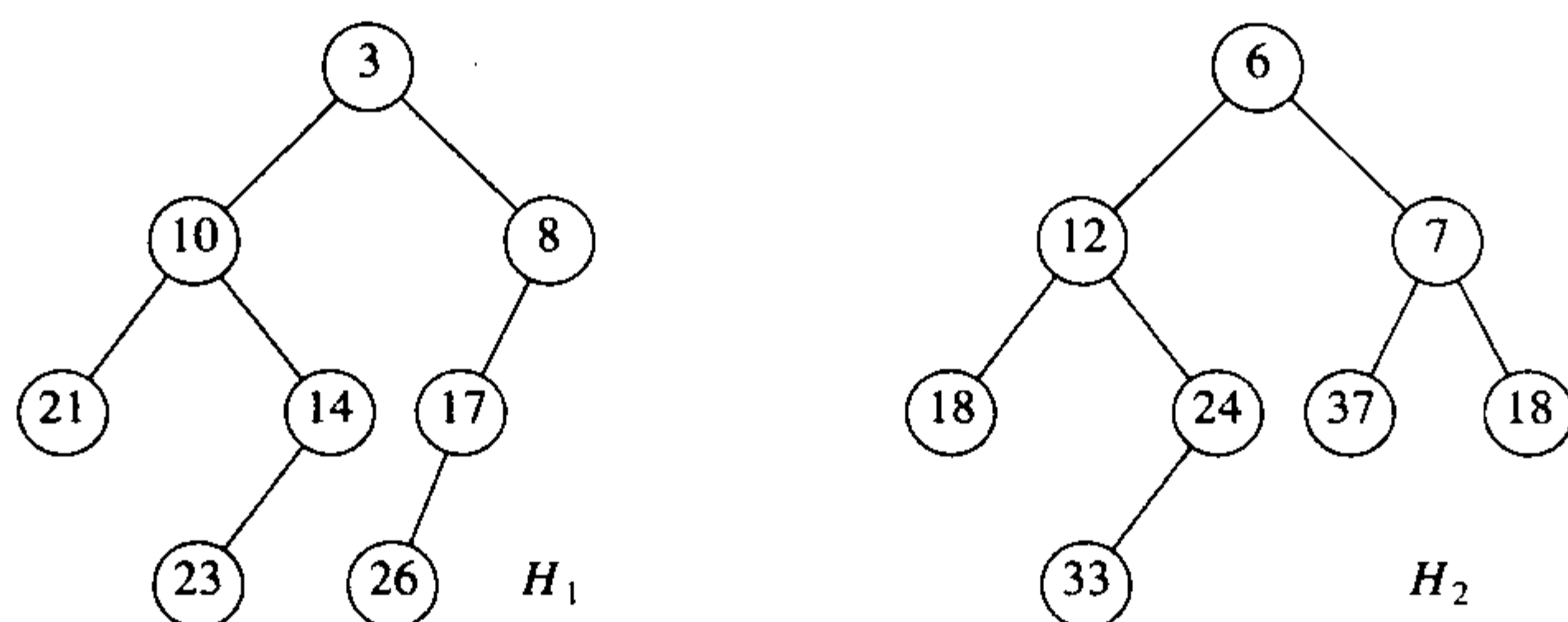
斜堆（skew heap）是左式堆的自调节形式，实现起来极其简单。斜堆和左式堆间的关系类似于伸展树和AVL树间的关系。斜堆是具有堆序的二叉树，但是不存在对树的结构限制。不同于左式堆，关于任意结点的零路径长的任何信息都不保留。斜堆的右路径在任何时刻都可以任意长，因此，所有操作的最坏情形运行时间均为 $O(N)$ 。然而，正如伸展树一样，可以证明（见第11章）对任意 $M$ 次连续操作，总的最坏情形运行时间是 $O(M \log N)$ 。因此，斜堆每次操作的摊还开销（amortized cost）为 $O(\log N)$ 。



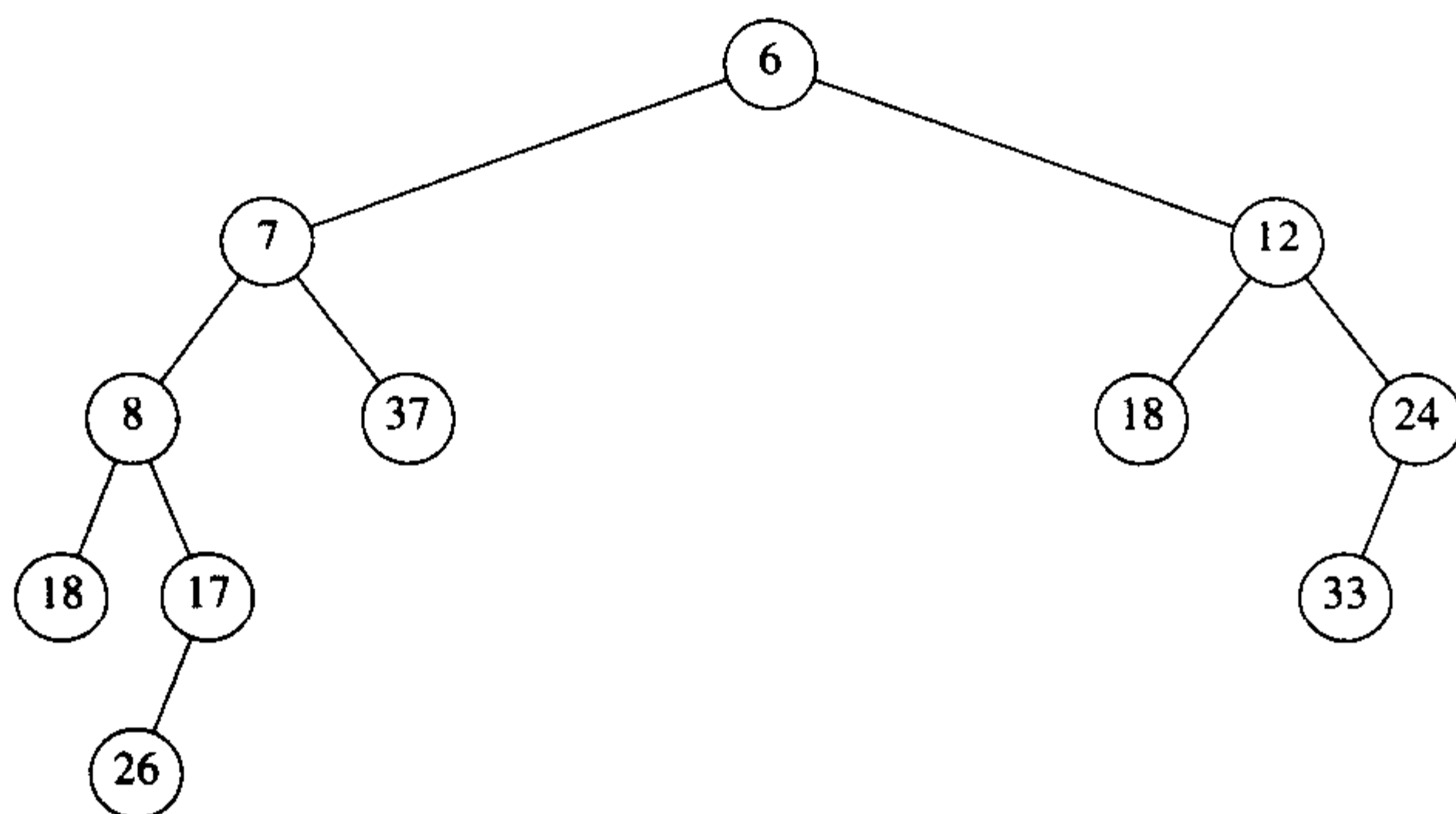
234  
235

与左式堆相同，斜堆的基本操作也是合并操作。merge例程还是递归的，我们执行与以前完全相同的操作，但有一个例外，即：对于左式堆，我们查看是否左儿子和右儿子满足左式堆的结构性质，并在不满足该性质时将它们交换。但对于斜堆，交换是无条件的，除右路径上所有结点的最大者不交换它的左右儿子外，我们都要进行这种交换。这个例外是在自然递归实现时所发生的现象，因此它实际上根本不是特殊情形。此外，证明时间界也是不必要的，但是，由于这样的结点肯定没有右儿子，因此执行交换是愚蠢的（在我们的例子中，该结点没有儿子，因此不必为此担心）。另外，仍设输入是与前面相同的两个堆，见图6-31。

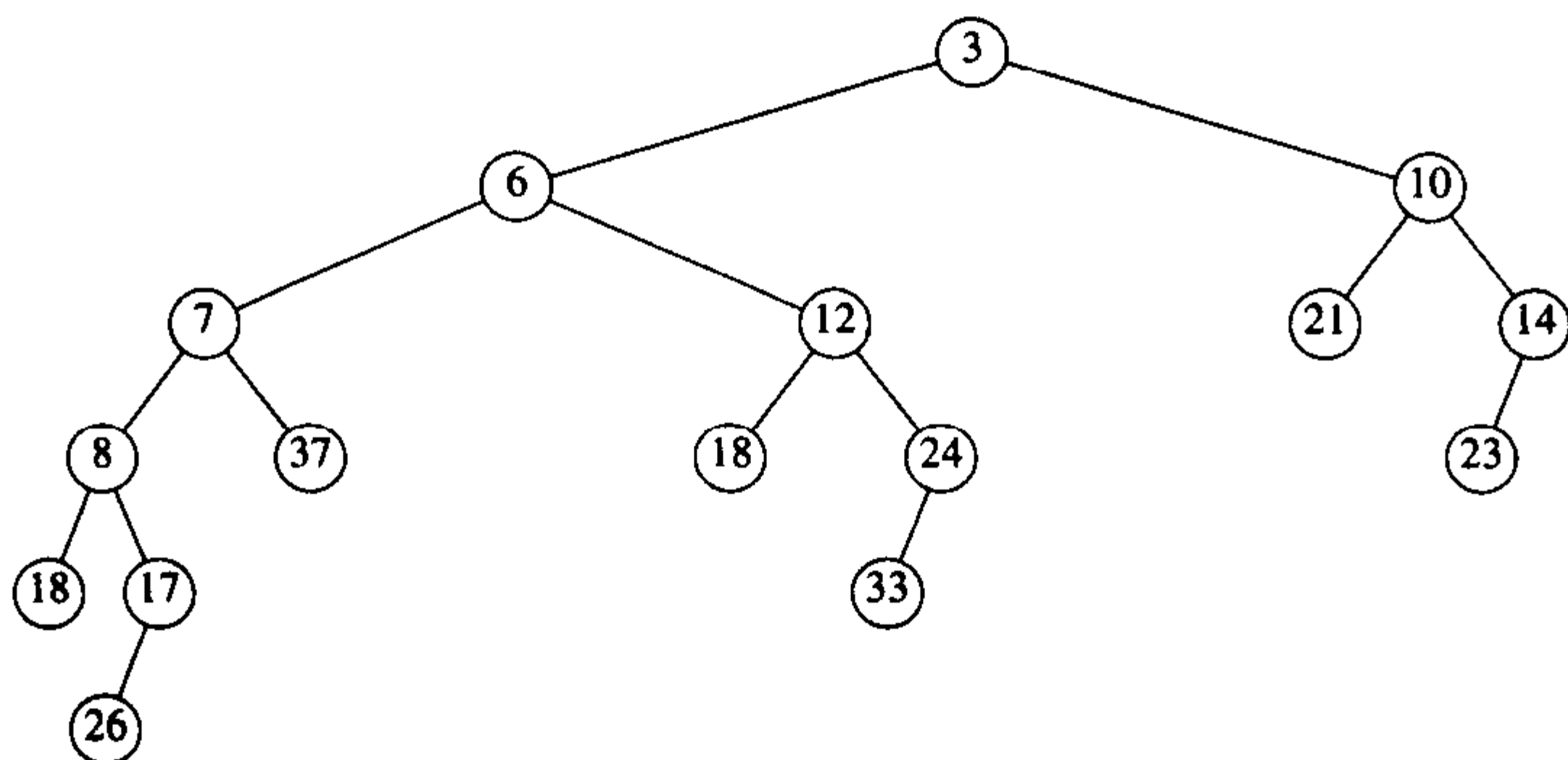
236

图6-31 两个斜堆 $H_1$ 和 $H_2$ 

如果我们递归地将 $H_2$ 与 $H_1$ 的根在8处的子堆合并，那么我们将得到图6-32中的堆。

图6-32 将 $H_2$ 与 $H_1$ 的右子堆合并的结果

这也是递归地完成的，因此，根据递归的第三个法则（见1.3节）我们不必担心它是如何得到的。这个堆碰巧是左式的，不过不能保证情况总是如此。我们使这个堆成为 $H_1$ 的新的左儿子，而 $H_1$ 的老的左儿子变成了新的右儿子（见图6-33）。

图6-33 合并斜堆 $H_1$ 和 $H_2$ 的结果

整个树是左式的，但是容易看到这并不总是成立的：将15插入到新堆中将破坏左式性质。

我们也可像左式堆那样非递归地进行所有的操作：合并右路径，除最后的结点外交换右路径上每个结点的左儿子和右儿子。经过几个例子之后，事情变得很清楚，由于除去右路径上最后的结点外的所有结点都将它们的儿子交换，因此最终效果是它变成了新的左路径（参见前面的例子以便使你自己确信）。这使得合并两个斜堆非常容易。<sup>1</sup>

238

斜堆的实现留做一个小练习。注意，因为右路径可能很长，所以递归实现可能由于缺乏栈空间而失败，不过在其他方面的性能是可接受的。斜堆有一个优点，即不需要附加的空间保留路径长以及不需要测试确定何时交换儿子。精确确定左式堆和斜堆的期望的右路径长是一个尚未解决的问题（后者无疑更为困难）。这样的比较将使得确定平衡信息的轻微遗失是否可由缺少测试来补偿更为容易。

## 6.8 二项队列

虽然左式堆和斜堆都以每次操作花费 $O(\log N)$ 时间有效地支持合并、插入和deleteMin，但还是有改进的余地，因为我们知道，二叉堆以每次操作花费常数时间支持插入。二项队列支持所有这三种操作，每次操作的最坏情形运行时间为 $O(\log N)$ ，而插入操作平均花费常数时间。

239

### 6.8.1 二项队列结构

二项队列（binomial queue）不同于前面介绍的所有优先队列的实现之处在于，一个二项队列不是一棵堆序的树，而是堆序的树的集合，称为森林（forest）。堆序树中的每一棵都是有约束的形式，叫作二项树（binomial tree）（后面将看到该名称的由来是显而易见的）。每一个高度上至多存在一棵二项树。高度为0的二项树是一棵单结点树；高度为 $k$ 的二项树 $B_k$ 通过将一棵二项树 $B_{k-1}$ 附接到另一棵二项树 $B_{k-1}$ 的根上而构成。图6-34显示二项树 $B_0$ 、 $B_1$ 、 $B_2$ 、 $B_3$ 以及 $B_4$ 。

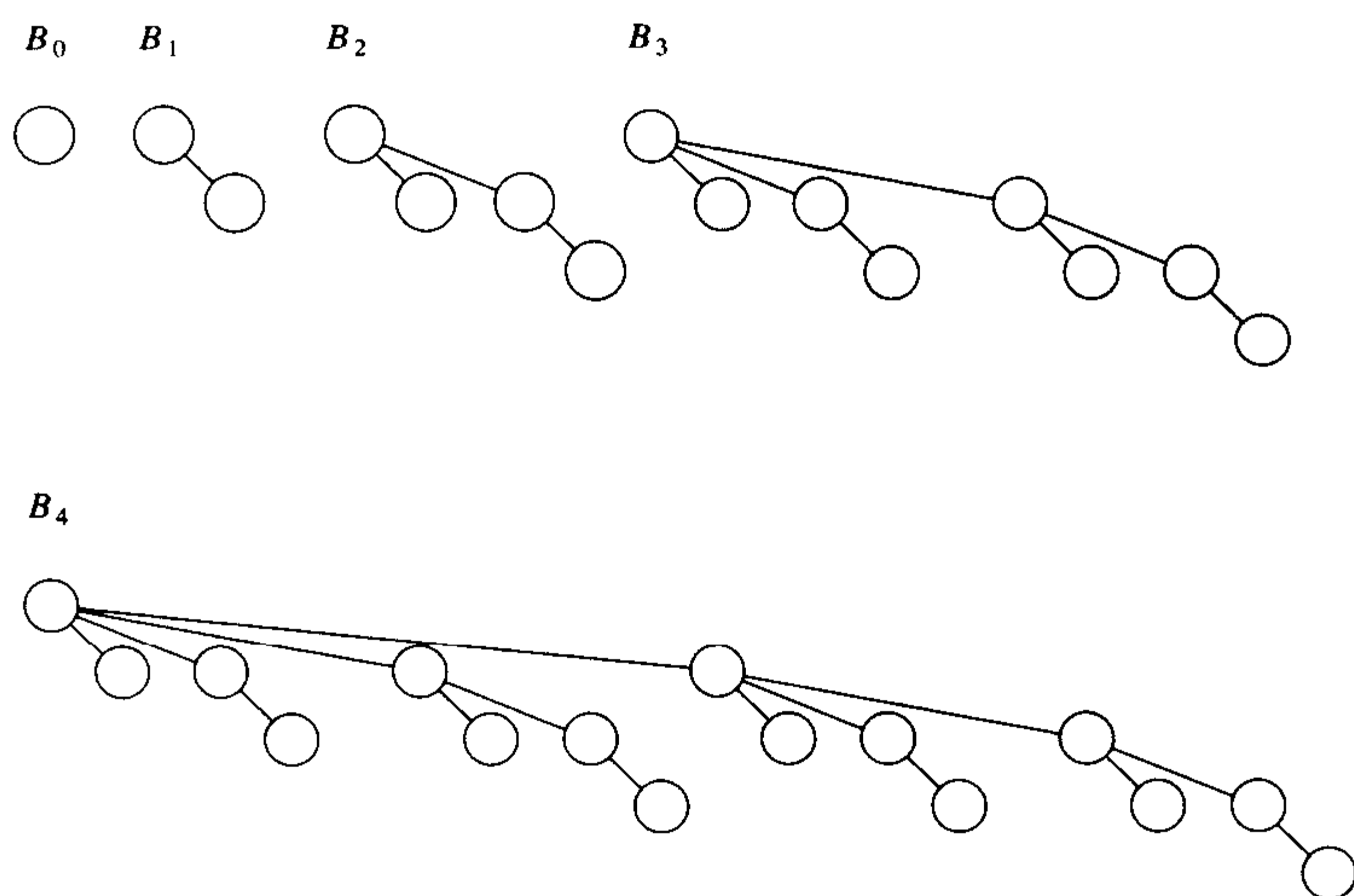


图6-34 二项树 $B_0$ 、 $B_1$ 、 $B_2$ 、 $B_3$ 以及 $B_4$

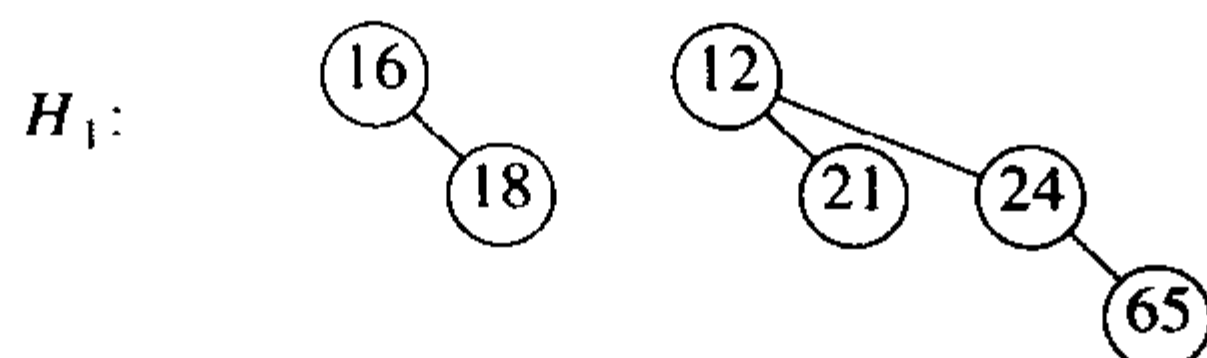
从图中看到，二项树 $B_k$ 由一个带有儿子 $B_0, B_1, \dots, B_{k-1}$ 的根组成。高度为 $k$ 的二项树恰好有 $2^k$

1. 这与递归实现不完全一样（但服从相同的时间界）。如果我们只交换由于一个堆的右路径用完而导致右路径合并终止的那一点上面的右路径上那些结点的儿子，那么我们将得到与递归做法相同的结果。

个结点，而在深度 $d$ 处的结点数是二项系数 $\binom{k}{d}$ 。如果我们把堆序施加到二项树上并允许任意高度上最多一棵二项树，那么我们能够用二项树的集合唯一地表示任意大小的优先队列。例如，大小为13的优先队列可以用森林 $B_3, B_2, B_0$ 表示。我们可以把这种表示写成1101，它不仅以二进制表示了13，而且也表示这样的事实：在上述表示中， $B_3, B_2, B_0$ 出现，而 $B_1$ 则没有出现。

240

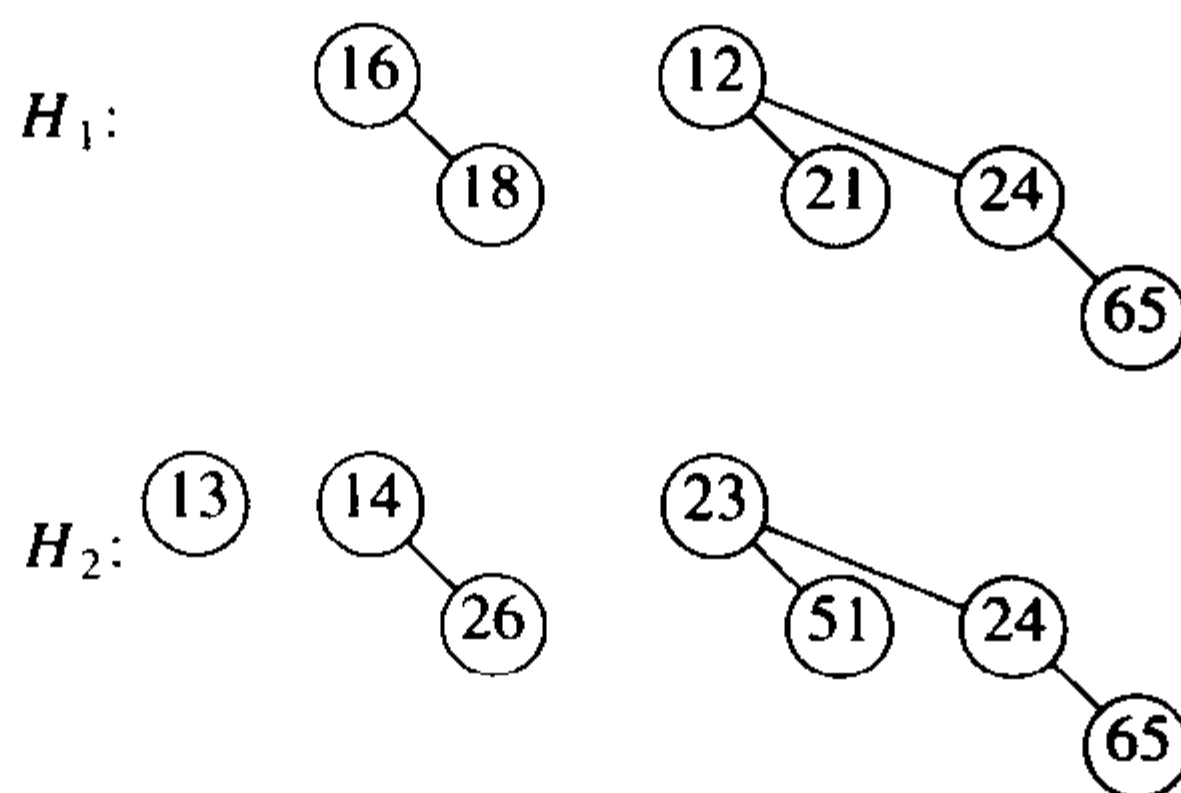
作为例子，6个元素的优先队列可以表示为图6-35中的形状。

图6-35 具有6个元素的二项队列 $H_1$ 

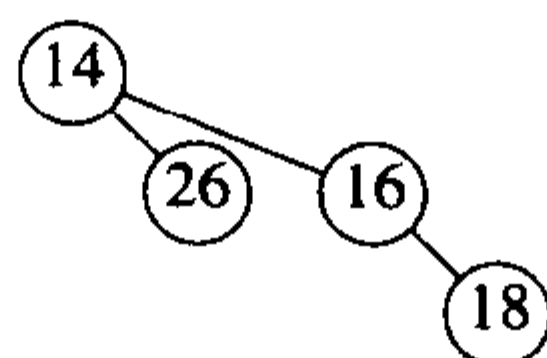
### 6.8.2 二项队列操作

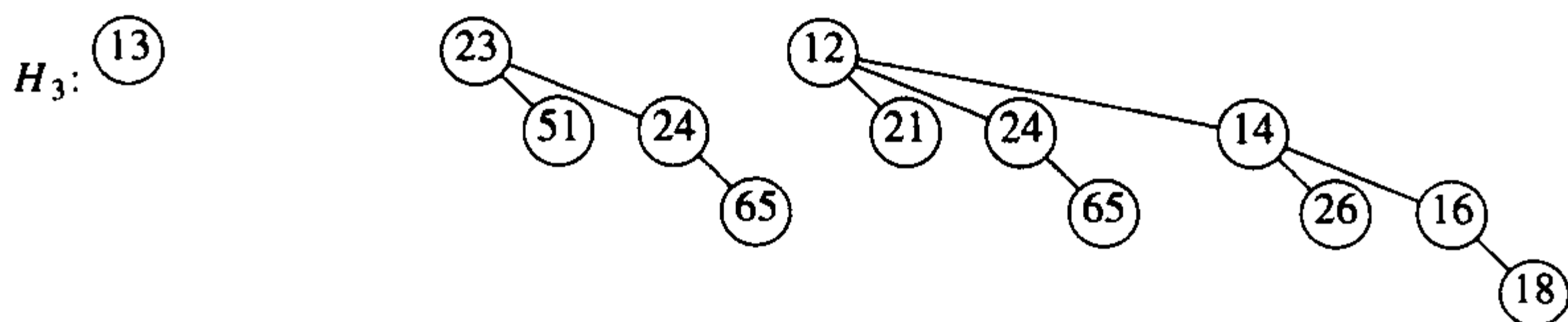
最小元可以通过搜索所有树的根来找出。由于最多有 $\log N$ 棵不同的树，因此最小元可以以 $O(\log N)$ 时间找到。另外，如果我们记住当最小元在其他操作期间变化时更新它，那么也可保留最小元的信息并以 $O(1)$ 时间执行该操作。

合并两个二项队列在概念上是一个容易的操作，我们将通过例子描述。考虑两个二项队列 $H_1$ 和 $H_2$ ，它们分别具有6个和7个元素，见图6-36。

图6-36 两个二项队列 $H_1$ 和 $H_2$ 

合并操作基本上是通过将两个队列加到一起完成的。令 $H_3$ 是新的二项队列。由于 $H_1$ 没有高度为0的二项树而 $H_2$ 有，因此我们就用 $H_2$ 中高度为0的二项树作为 $H_3$ 的一部分。然后，将两个高度为1的二项树相加。由于 $H_1$ 和 $H_2$ 都有高度为1的二项树，因此可以将它们合并，让大的根成为小的根的子树，从而建立高度为2的二项树，见图6-37。这样， $H_3$ 将没有高度为1的二项树。现在存在三棵高度为2的二项树，即 $H_1$ 和 $H_2$ 原有的两个二项树以及由上一步形成的一棵二项树。我们将一棵高度为2的二项树放到 $H_3$ 中并合并其他两个二项树，得到一棵高度为3的二项树。由于 $H_1$ 和 $H_2$ 都没有高度为3的二项树，因此该二项树就成为 $H_3$ 的一部分，合并结束。最后得到的二项队列如图6-38所示。

图6-37  $H_1$ 和 $H_2$ 中两棵 $B_1$ 树合并

图6-38 二项队列 $H_3$ : 合并 $H_1$ 和 $H_2$ 的结果

由于几乎使用任意合理的实现方法合并两棵二项树均花费常数时间, 而总共存在 $O(\log N)$ 棵二项树, 因此合并在最坏情形下花费时间为 $O(\log N)$ 。为使该操作更有效, 我们需要将这些树放到按照高度排序的二项队列中, 当然这做起来是件简单的事情。

241

插入实际上就是特殊情形的合并, 只要创建一棵单结点树并执行一次合并即可。这种操作的最坏情形运行时间也是 $O(\log N)$ 。更准确地说, 如果元素将要插入的那个优先队列中不存在的最小的二项树是 $B_i$ , 那么运行时间与 $i+1$ 成正比。例如,  $H_3$  (见图6-38) 缺少高度为1的二项树, 因此插入将进行两步而终止。由于二项队列中每棵树出现的概率均为 $1/2$ , 于是我们预计插入在两步后终止, 因此, 平均时间是常数。不仅如此, 分析将指出, 对一个初始为空的二项队列进行 $N$ 次insert将花费 $O(N)$ 最坏情形时间。事实上, 只用 $N-1$ 次比较就有可能进行该操作; 我们把它留做练习。

作为一个例子, 我们用图6-39到图6-45演示通过依序插入1~7来构成一个二项队列。4的插入展现一种不好的情形。我们把4与 $B_0$ 合并, 得到一棵新的高度为1的树。然后将该树与 $B_1$ 合并, 得到一棵高度为2的树, 它是新的优先队列。我们把这些算做三步 (两次树合并加上终止情形)。在插入7以后的下一次插入又是一个不好情形, 需要三次树合并操作。

242



图6-39 在1插入之后

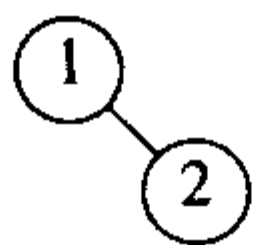


图6-40 在2插入之后

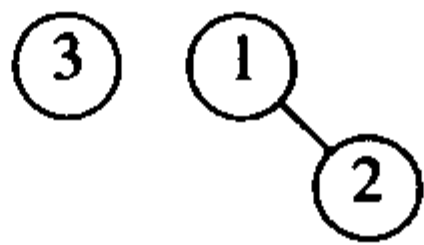


图6-41 在3插入之后

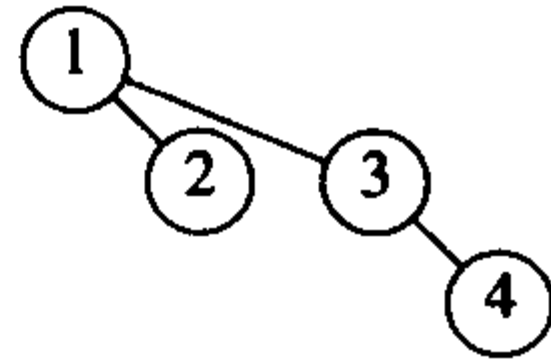


图6-42 在4插入之后

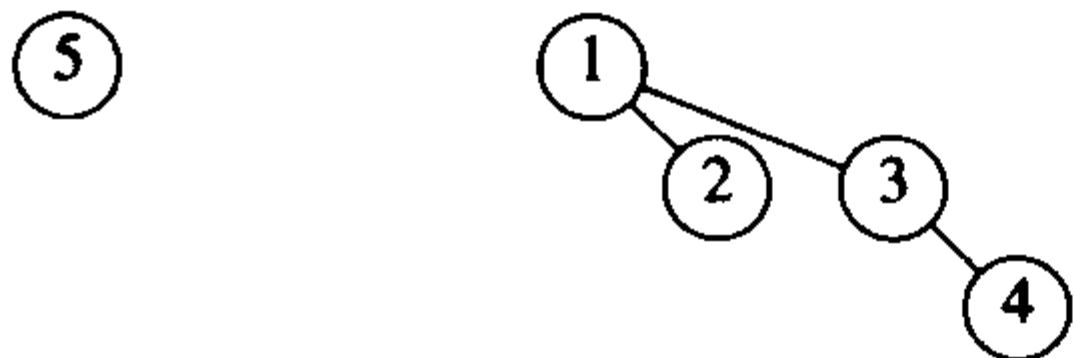


图6-43 在5插入之后

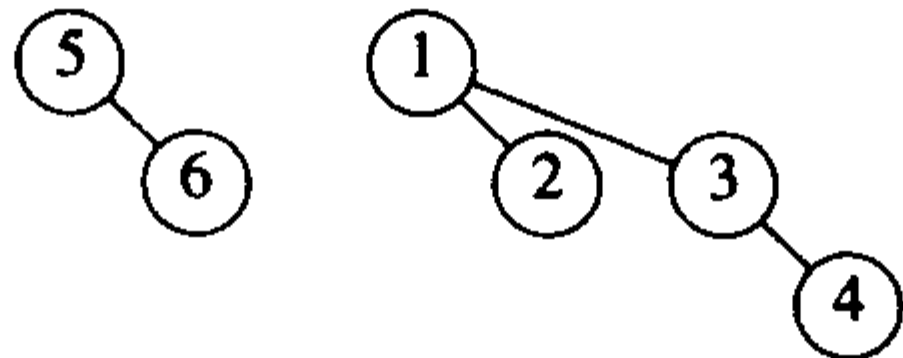


图6-44 在6插入之后

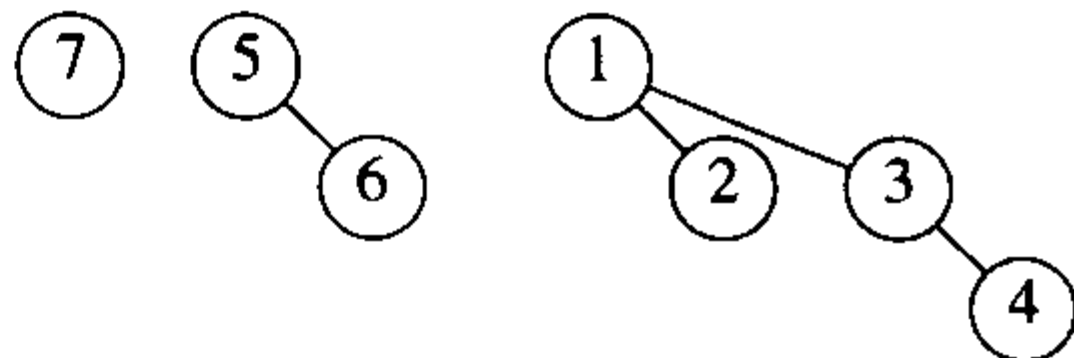
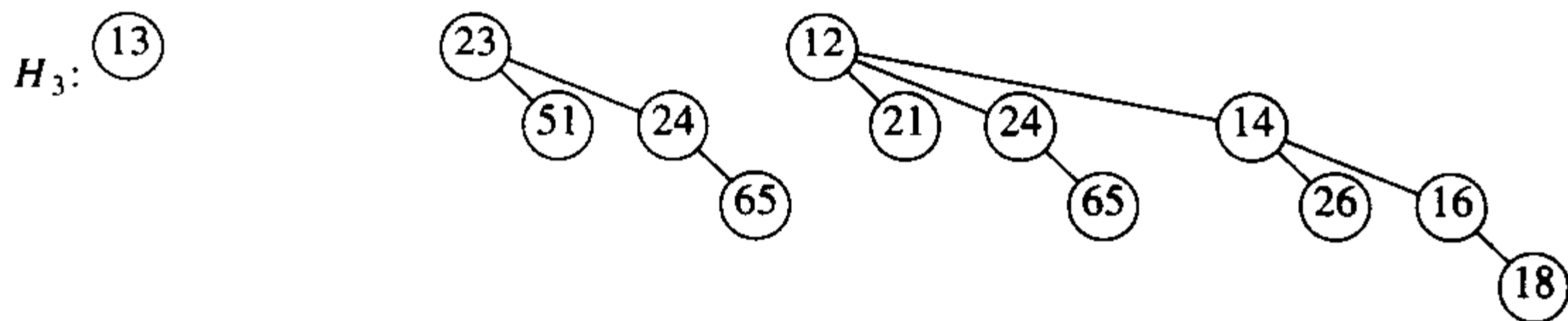


图6-45 在7插入之后

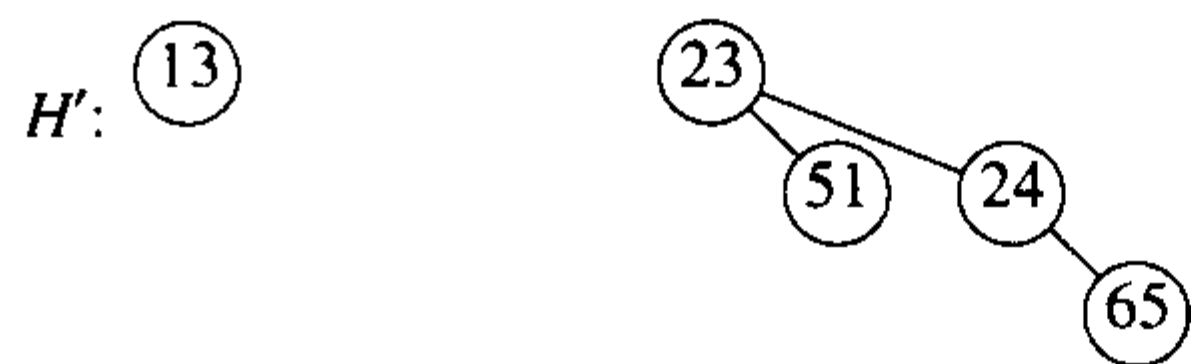
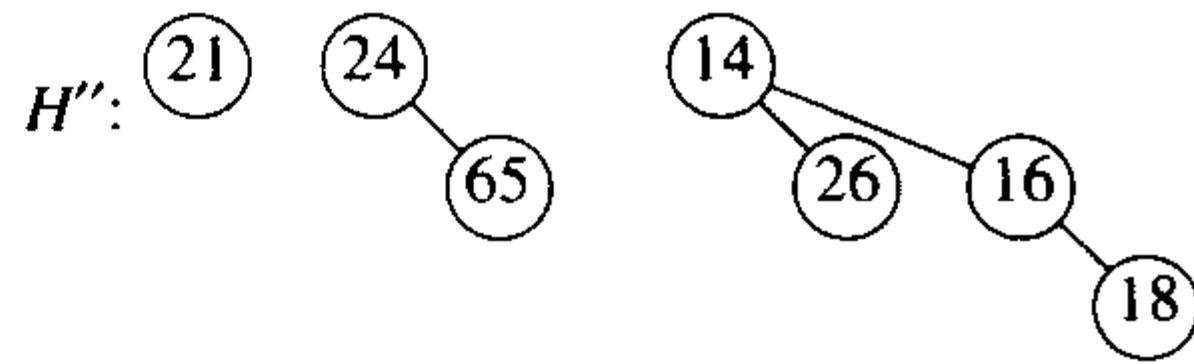
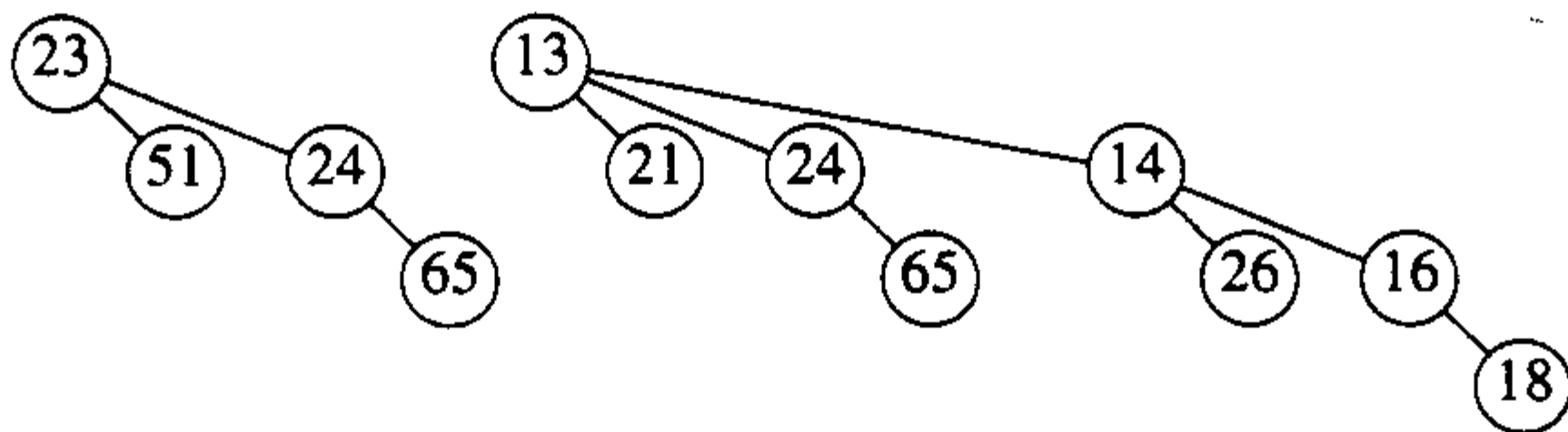
deleteMin可以通过首先找出一棵具有最小根的二项树来完成。令该树为 $B_k$ , 并令原始的优先队列为 $H$ 。我们从 $H$ 的树的森林中除去二项树 $B_k$ , 形成新的二项树队列 $H'$ 。再除去 $B_k$ 的根, 得到一些二项树 $B_0, B_1, \dots, B_{k-1}$ , 它们共同形成优先队列 $H''$ 。合并 $H'$ 和 $H''$ , 操作结束。

作为例子, 设对 $H_3$ 执行一次deleteMin, 它也在图6-46中表示出。最小的根是12, 因此我们得到图6-47和图6-48中的两个优先队列 $H'$ 和 $H''$ 。合并 $H'$ 和 $H''$ 得到的二项队列是最后的答案, 见图6-49所示。

243

图6-46 二项队列 $H_3$



图6-47 二项队列 $H'$ ，包含除 $B_3$ 外的 $H_3$ 中所有的二项树图6-48 二项队列 $H''$ ：除去12后的 $B_3$ 图6-49 deleteMin应用到 $H_3$ 的结果

为了分析，首先注意，deleteMin操作将原二项队列一分为二。找出含有最小元素的树并创建队列 $H$ 和 $H'$ 花费 $O(\log N)$ 时间。合并这两个队列又花费 $O(\log N)$ 时间，因此，整个deleteMin操作花费 $O(\log N)$ 时间。

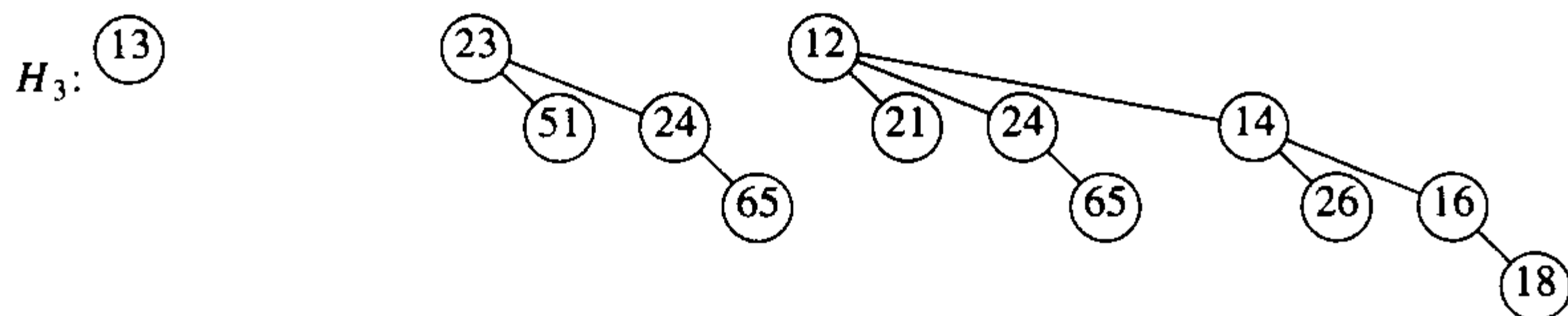
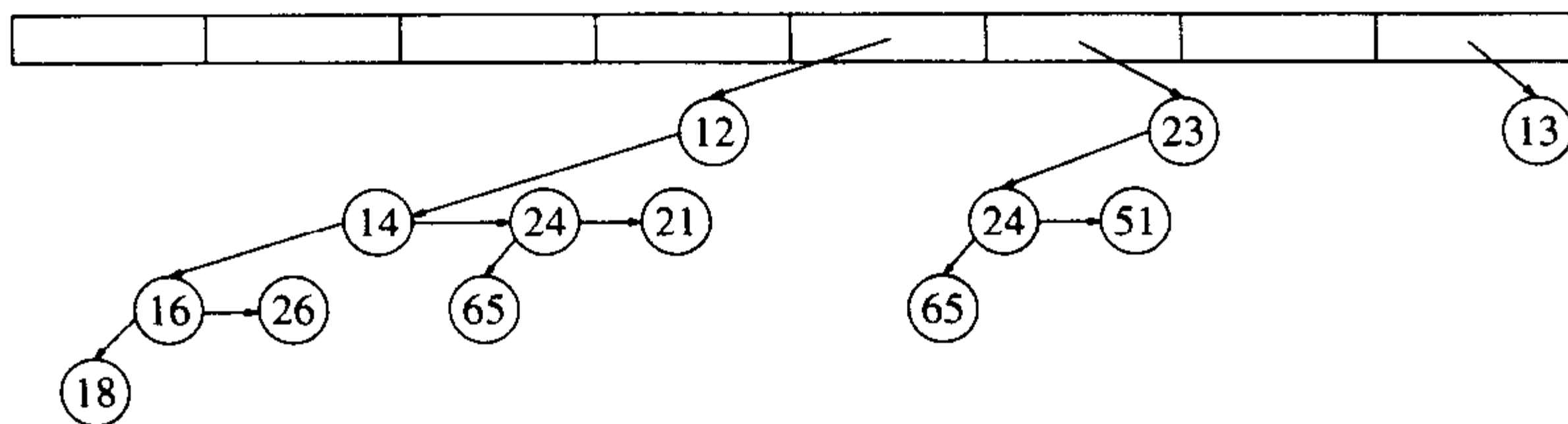
### 6.8.3 二项队列的实现

**244** deleteMin操作需要快速找出根的所有子树的能力，因此，需要一般树的标准表示方法：每个结点的儿子都在一个链表中，而且每个结点都有一个指向它的第一个儿子（如果有的话）的指针。该操作还要求诸儿子按照它们的子树的大小排序。我们还需要保证很容易合并两棵树。当两棵树被合并时，其中的一棵树作为儿子加到另一棵树上。由于这棵新树将是最大的子树，因此，以大小递减的方式保持这些子树是有意义的。只有这时我们才能够有效地合并两棵二项树从而合并两个二项队列。二项队列将是二项树的数组。

总之，二项树的每一个结点将包含数据、第一个儿子以及右兄弟。二项树中的儿子以递减次序排列。

图6-51说明了如何表示图6-50中的二项队列。图6-52显示了二项树中结点的类型声明以及二项队列的类架构。

**245**

图6-50 画成森林的二项队列 $H_3$ 图6-51 二项队列 $H_3$ 的表示方式

```

1  template <typename Comparable>
2  class BinomialQueue
3  {
4  public:
5      BinomialQueue( );
6      BinomialQueue( const Comparable & item );
7      BinomialQueue( const BinomialQueue & rhs );
8      ~BinomialQueue( );
9
10     bool isEmpty( ) const;
11     const Comparable & findMin( ) const;
12
13     void insert( const Comparable & x );
14     void deleteMin( );
15     void deleteMin( Comparable & minItem );
16
17     void makeEmpty( );
18     void merge( BinomialQueue & rhs );
19
20     const BinomialQueue & operator= ( const BinomialQueue & rhs );
21
22 private:
23     struct BinomialNode
24     {
25         Comparable    element;
26         BinomialNode *leftChild;
27         BinomialNode *nextSibling;
28
29         BinomialNode( const Comparable & theElement,
30                     BinomialNode *lt, BinomialNode *rt )
31             : element( theElement ), leftChild( lt ), nextSibling( rt ) { }
32     };
33
34     enum { DEFAULT_TREES = 1 };
35
36     int currentSize;                // Number of items in priority queue
37     vector<BinomialNode *> theTrees; // An array of tree roots
38
39     int findMinIndex( ) const;
40     int capacity( ) const;
41     BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 );
42     void makeEmpty( BinomialNode * & t );
43     BinomialNode * clone( BinomialNode *t ) const;
44 };

```

图6-52 二项队列类构架及结点定义

为了合并两个二项队列，我们需要一个例程来合并两棵同样大小的二项树。图6-53表明，两棵二项树合并时链是如何变化的。合并二项树的程序很简单，参见图6-54。

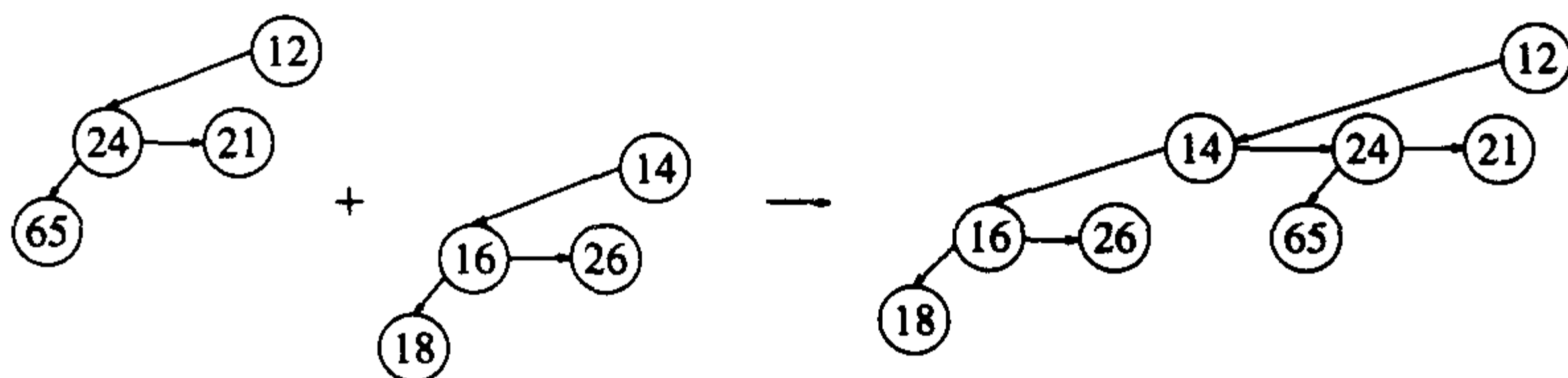


图6-53 合并两棵二项树

```

1 /**
2  * Return the result of merging equal-sized t1 and t2.
3  */
4 BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 )
5 {
6     if( t2->element < t1->element )
7         return combineTrees( t2, t1 );
8     t2->nextSibling = t1->leftChild;
9     t1->leftChild = t2;
10    return t1;
11 }

```

图6-54 合并同样大小的两棵二项树的例程

下面介绍merge例程的简单实现。 $H_1$ 由当前的对象表示，而 $H_2$ 则用rhs表示。该例程将 $H_1$ 和 $H_2$ 合并，把合并结果放入 $H_1$ 中，并清空 $H_2$ 。在任意时刻我们在处理的是秩为 $i$ 的那些树。 $t_1$ 和 $t_2$ 分别是 $H_1$ 和 $H_2$ 中的树，而carry是从上一步得来的树（可能是NULL）。从秩为 $i$ 的树以及秩为 $i+1$ 的carry树所形成的树，其形成过程依赖于8种可能情形中的每一种。该过程从秩0开始直到产生二项队列的最后的秩。程序见图6-55。代码的改进见练习6.35。

```

1 /**
2  * Merge rhs into the priority queue.
3  * rhs becomes empty. rhs must be different from this.
4  */
5 void merge( BinomialQueue & rhs )
6 {
7     if( this == &rhs )    // Avoid aliasing problems
8         return;
9
10    currentSize += rhs.currentSize;
11
12    if( currentSize > capacity( ) )
13    {
14        int oldNumTrees = theTrees.size( );
15        int newNumTrees = max( theTrees.size( ), rhs.theTrees.size( ) ) + 1;
16        theTrees.resize( newNumTrees );
17        for( int i = oldNumTrees; i < newNumTrees; i++ )
18            theTrees[ i ] = NULL;
19    }
20
21    BinomialNode *carry = NULL;
22    for( int i = 0, j = 1; j <= currentSize; i++, j *= 2 )
23    {
24        BinomialNode *t1 = theTrees[ i ];
25        BinomialNode *t2 = i < rhs.theTrees.size( ) ? rhs.theTrees[ i ]
26                                : NULL;
27        int whichCase = t1 == NULL ? 0 : 1;
28        whichCase += t2 == NULL ? 0 : 2;
29        whichCase += carry == NULL ? 0 : 4;
30
31        switch( whichCase )
32        {
33            case 0: /* No trees */
34            case 1: /* Only this */
35                break;
36            case 2: /* Only rhs */
37                theTrees[ i ] = t2;

```

图6-55 合并两个优先队列的例程

```

38         rhs.theTrees[ i ] = NULL;
39         break;
40     case 4: /* Only carry */
41         theTrees[ i ] = carry;
42         carry = NULL;
43         break;
44     case 3: /* this and rhs */
45         carry = combineTrees( t1, t2 );
46         theTrees[ i ] = rhs.theTrees[ i ] = NULL;
47         break;
48     case 5: /* this and carry */
49         carry = combineTrees( t1, carry );
50         theTrees[ i ] = NULL;
51         break;
52     case 6: /* rhs and carry */
53         carry = combineTrees( t2, carry );
54         rhs.theTrees[ i ] = NULL;
55         break;
56     case 7: /* All three */
57         theTrees[ i ] = carry;
58         carry = combineTrees( t1, t2 );
59         rhs.theTrees[ i ] = NULL;
60         break;
61     }
62 }
63
64 for( int k = 0; k < rhs.theTrees.size( ); k++ )
65     rhs.theTrees[ k ] = NULL;
66 rhs.currentSize = 0;
67 }

```

图6-55 合并两个优先队列的例程（续）

二项队列的deleteMin例程在图6-56中给出。

当受到影响的元素的位置已知时，我们可以将二项队列扩展到支持二叉堆所允许的某些非标准的操作，诸如decreaseKey和remove等。decreaseKey是一次percolateUp，如果我们将一个字段加到每个存储其父链的结点上，那么这个操作可以以 $O(\log N)$ 时间完成。一次任意的remove可以通过结合decreaseKey和deleteMin而以 $O(\log N)$ 时间完成。

246  
250

## 6.9 标准库中的优先队列

在STL中，二叉堆是通过称为priority\_queue的类模板实现的，该类模板可以在标准头文件queue中找到。STL实现一个最大堆（max-heap）而不是最小堆（min-heap），于是所访问的项就是最大的项而不是最小的项。键成员函数如下：

```

void push( const Object & x );
const Object & top( ) const;
void pop( );
bool empty( );
void clear( );

```

push将x添加到优先队列中，top返回优先队列中的最大项，而pop删除优先队列中的最大项。重复是允许的。如果有数个最大元素，只有其中的一个被删除掉。

优先队列模板用如下参数初始化：项类型、容器类型（几乎总是项使用存储项的vector）和比较器。最后两个参数默认值是使用的。并且默认情况下得到最大堆。使用greater函数对



象作为比较器可以得到最小堆。

图6-57是一个测试程序。该程序表示priority\_queue类模板在默认值为最大堆和最小堆的两种情况下的使用。

```

1  /**
2   * Remove the minimum item and place it in minItem.
3   * Throws UnderflowException if empty.
4   */
5  void deleteMin( Comparable & minItem )
6  {
7      if( isEmpty( ) )
8          throw UnderflowException( );
9
10     int minIndex = findMinIndex( );
11     minItem = theTrees[ minIndex ]->element;
12
13     BinomialNode *oldRoot = theTrees[ minIndex ];
14     BinomialNode *deletedTree = oldRoot->leftChild;
15     delete oldRoot;
16
17     // Construct H''
18     BinomialQueue deletedQueue;
19     deletedQueue.theTrees.resize( minIndex + 1 );
20     deletedQueue.currentSize = ( 1 << minIndex ) - 1;
21     for( int j = minIndex - 1; j >= 0; j-- )
22     {
23         deletedQueue.theTrees[ j ] = deletedTree;
24         deletedTree = deletedTree->nextSibling;
25         deletedQueue.theTrees[ j ]->nextSibling = NULL;
26     }
27
28     // Construct H'
29     theTrees[ minIndex ] = NULL;
30     currentSize -= deletedQueue.currentSize + 1;
31
32     merge( deletedQueue );
33 }
34
35 /**
36 * Find index of tree containing the smallest item in the priority queue.
37 * The priority queue must not be empty.
38 * Return the index of tree containing the smallest item.
39 */
40 int findMinIndex( ) const
41 {
42     int i;
43     int minIndex;
44
45     for( i = 0; theTrees[ i ] == NULL; i++ )
46         ;
47
48     for( minIndex = i; i < theTrees.size( ); i++ )
49         if( theTrees[ i ] != NULL &&
50             theTrees[ i ]->element < theTrees[ minIndex ]->element )
51             minIndex = i;
52
53     return minIndex;
54 }

```

图6-56 二项队列的deleteMin

```

1 #include <iostream>
2 #include <vector>
3 #include <queue>
4 #include <functional>
5 #include <string>
6 using namespace std;
7
8 // Empty the priority queue and print its contents.
9 template <typename PriorityQueue>
10 void dumpContents( const string & msg, PriorityQueue & pq )
11 {
12     cout << msg << ":" << endl;
13     while( !pq.empty( ) )
14     {
15         cout << pq.top( ) << endl;
16         pq.pop( );
17     }
18 }
19
20 // Do some inserts and removes (done in dumpContents).
21 int main( )
22 {
23     priority_queue<int>                                maxPQ;
24     priority_queue<int,vector<int>,greater<int> > minPQ;
25
26     minPQ.push( 4 ); minPQ.push( 3 ); minPQ.push( 5 );
27     maxPQ.push( 4 ); maxPQ.push( 3 ); maxPQ.push( 5 );
28
29     dumpContents( "minPQ", minPQ );    // 3 4 5
30     dumpContents( "maxPQ", maxPQ );    // 5 4 3
31
32     return 0;
33 }

```

图6-57 STL的priority\_queue例程；注释说明了期望的输出顺序

## 小结

在本章，我们已经看到优先队列ADT的各种实现方法和用途。标准的二叉堆实现由于简单和速度快从而很雅致。它不需要链，只需要常量的附加空间，且有效地支持优先队列的操作。

我们考虑了附加的merge操作，开发了三种实现方法，每种都有其独到之处。左式堆是递归能力的完美实例。斜堆则代表缺少平衡原则的一种重要的数据结构。它的分析是有趣的，我们将在第11章进行。二项队列表现出，一个简单的想法如何用来达到好的时间界。

我们还看到优先队列的几个用途，从操作系统的工作调度到事件模拟。我们将在第7章、第9章和第10章再次看到它们的应用。

## 练习

- 6.1 操作insert和操作findMin都能以常数时间实现吗？
- 6.2 a. 写出一次一个地将10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13和2插入到一个初始为空的二叉堆中的结果。
- b. 写出使用相同的输入通过线性时间算法建立一个二叉堆的结果。

- 6.3 写出对上面练习中的堆执行3次deleteMin操作的结果。
- 6.4  $N$ 个元素的完全二叉树用到数组位置1到 $N$ 。设我们试图使用数组表示法表示非完全的二叉树。对于下列的情况确定数组必须要多大：
- 一棵有两个附加层（即非常轻微地不平衡）的二叉树。
  - 在深度 $2 \log N$ 处有一个最深的结点的二叉树。
  - 在深度 $4.1 \log N$ 处有一个最深的结点的二叉树。
  - 最坏情形的二叉树。
- 6.5 重写BinaryHeap insert例程，在位置0处放置所插入项的一个副本。
- 6.6 在图6-13的大的堆中有多少结点？
- 6.7
- 证明对于二叉堆，buildHeap至多在元素间进行 $2N - 2$ 次比较。
  - 证明8个元素的堆可以通过堆元素间的8次比较构成。
  - \*\*c. 给出一个算法，使以 $\frac{13}{8}N + O(\log N)$ 次元素比较构建一个二叉堆。
- 6.8 证明下列关于堆中的最大项的结论：
- 它必须在一片树叶上。
  - 恰好存在 $\lceil N/2 \rceil$ 片树叶。
  - 为找出它，必须考查每一片树叶。
- \*\*6.9 证明，在一个大的完全堆（你可以假设 $N = 2^k - 1$ ）中第 $k$ 个最小元的期望深度以 $\log k$ 为界。
- 6.10
- \*a. 给出一个算法以找出二叉堆中小于某个值 $X$ 的所有结点。你的算法应该以 $O(K)$ 运行，其中， $K$ 是输出的结点的个数。
  - 你的算法可以扩展到本章讨论过的任何其他堆结构吗？
  - \*c. 给出一个算法，最多使用大约 $3N/4$ 次比较找出二叉堆中任意的项 $X$ 。
- \*\*6.11 提出一个算法，使以 $O(M + \log N \log \log N)$ 时间将 $M$ 个结点插入到 $N$ 个元素的二叉堆中。证明你的时间界。
- 6.12 编写一个程序输入 $N$ 个元素并：
- 将它们一个一个地插入到一个堆中。
  - 以线性时间建立一个堆。
- 比较这两个算法对于已排序、反序以及随机输入的运行时间。
- 6.13 每个deleteMin操作在最坏情形下使用 $2 \log N$ 次比较。
- \*a. 提出一种方案使得deleteMin操作只使用 $\log N + \log \log N + O(1)$ 次元素间的比较。这并不意味着较少的数据移动。
  - \*\*b. 扩展你在a部分中的方案使得只执行 $\log N + \log \log \log N + O(1)$ 次比较。
  - \*\*c. 你能够把这种想法推向多远？
  - d. 在比较中节省下的资源能否补偿你的算法增加的复杂性？
- 6.14 如果一个 $d$ 堆作为一个数组存储，对位于位置 $i$ 的项，其父亲和儿子都在哪里？
- 6.15 设一个 $d$ 堆初始时有 $N$ 个元素，而我们需要对其执行 $M$ 次percolateUp和 $N$ 次deleteMin。
- a. 用 $M$ 、 $N$ 和 $d$ 表示的所有操作的总运行时间是多少？
  - b. 如果 $d = 2$ ，所有的堆操作的运行时间是多少？
  - c. 如果 $d = \Theta(N)$ ，总的运行时间是多少？
  - \*d. 怎样选择 $d$ 将使总的运行时间最小？
- 6.16 设二叉堆用显式链表示。给出一个简单算法来找出位于位置 $i$ 上的树结点。
- 6.17 设二项堆用显式链表示。考虑将二叉堆lhs和rhs合并的问题。假设这两个二叉堆均为满的完全树，分别包含 $2^l - 1$ 和 $2^r - 1$ 个结点。
- a. 若 $l = r$ ，则给出合并这两个堆的 $O(\log N)$ 算法。

b. 若 $|l-r|=1$ , 则给出合并这两个堆的 $O(\log N)$ 算法。

c. 给出合并这两个堆的与 $l$ 和 $r$ 无关的 $O(\log^2 N)$ 算法。

6.18 最小—最大堆 (min-max heap) 是支持两种操作deleteMin和deleteMax的数据结构, 每个操作作用时 $O(\log N)$ 。该结构与二叉堆相同, 不过, 其堆序性质为: 对于在偶数深度上的任意结点 $X$ , 存储在 $X$ 上的元素小于它的父亲但是大于它的祖父 (当这是有意义的时候), 对于奇数深度上的任意结点 $X$ , 存储在 $X$ 上的元素大于它的父亲但是小于它的祖父, 见图6-58。

a. 如何找到最小元和最大元?

\*b. 给出一个算法将一个新结点插入到该最小—最大堆中。

\*c. 给出一个算法执行deleteMin和deleteMax。

\*d. 你能否以线性时间建立一个最小—最大堆?

\*\*e. 设我们想要支持操作deleteMin、deleteMax以及merge。提出一种数据结构以时间 $O(\log N)$ 支持所有的操作。

254

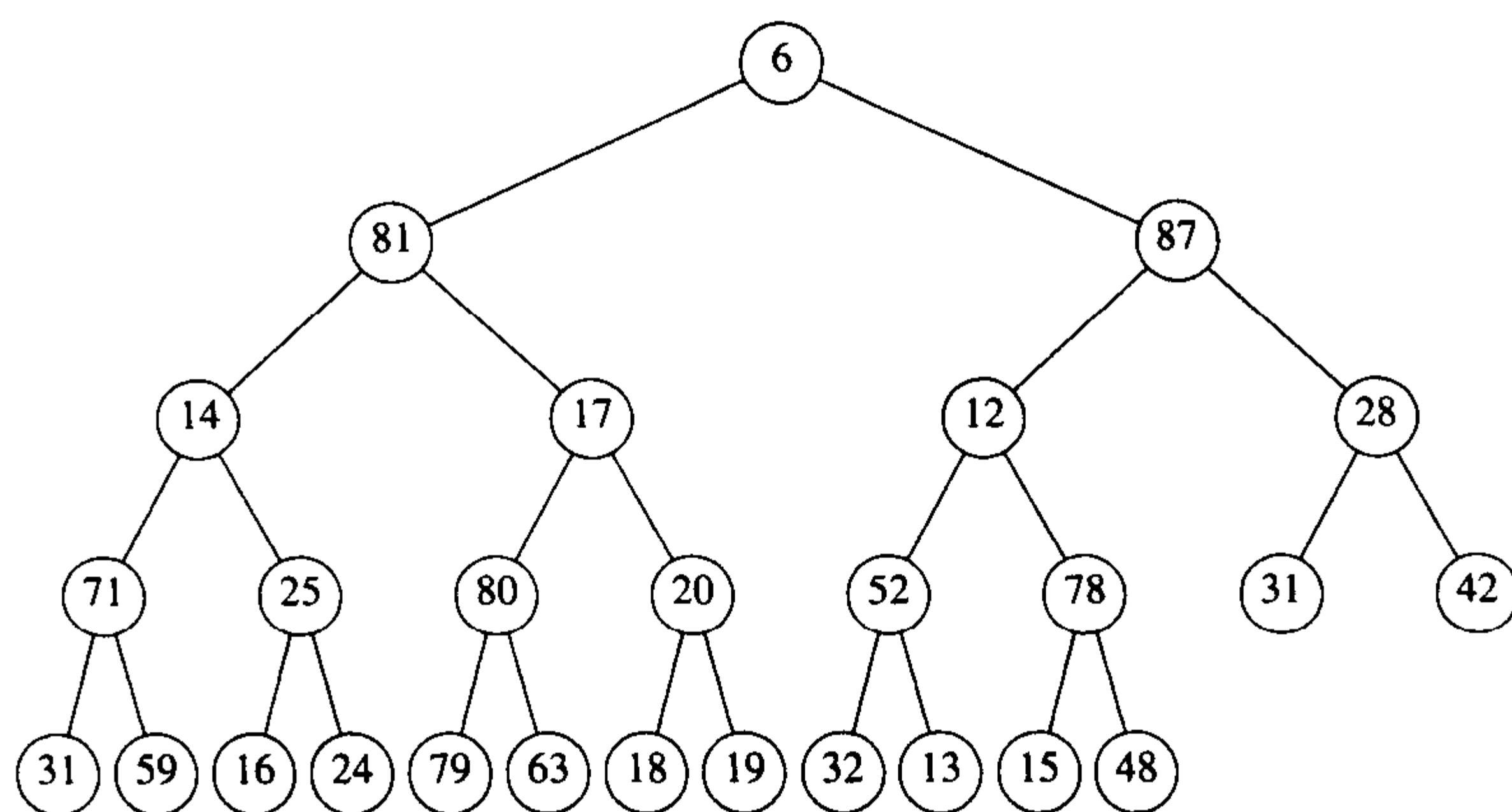


图6-58 最小—最大堆

6.19 合并图6-59中的两个左式堆。

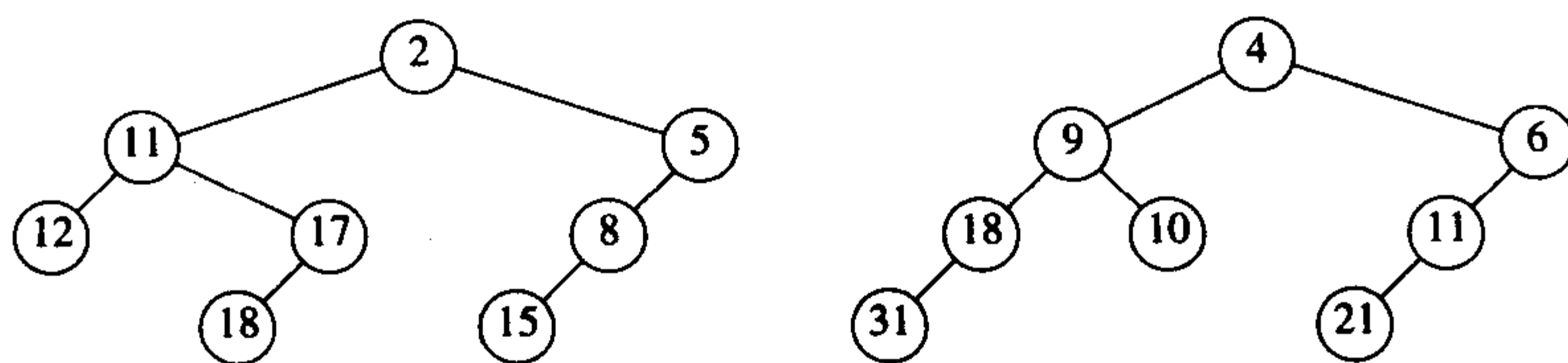


图6-59 练习6.19和练习6.26的输入

6.20 写出依序将键1到15插入一个初始为空的左式堆中的结果。

6.21 证明下述结论成立或证明其不成立: 如果将键1到 $2^k - 1$ 依序插入到一个初始为空的左式堆中, 那么结果形成一棵理想平衡树。

6.22 给出生成最佳左式堆的输入的例子。

6.23 a. 左式堆能否有效地支持decreaseKey?

b. 完成该功能需要哪些变化 (如果可能的话)?

6.24 从左式堆中一个已知位置删除结点的一种方法是使用懒惰策略。要删除一个结点, 只需将其标记为删除即可。当执行一个findMin或deleteMin时, 若根结点被标记“删除”则存在一个潜在的问题, 因为此时结点必须被实际删除且需要找到实际的最小元, 这可能涉及删除其他一些已做



标记的结点。在该策略中，这些remove花费一个单位，但一次deleteMin或findMin的开销却依赖于被做“删除”标记的结点的个数。设在一次deleteMin或findMin后做标记的结点比操作前少了 $k$ 个。

\*a. 说明如何以 $O(k \log N)$ 时间执行deleteMin。

\*\*b. 提出一种实现方法，通过分析证明执行deleteMin的时间为 $O(k \log (2N/k))$ 。

6.25 我们可以以线性时间对一些左式堆执行buildHeap操作：把每个元素当做是单结点左式堆，把所有这些堆放到一个队列中，之后，让两个堆出队，合并它们，再将合并结果入队，直到队列中只有一个堆为止。

a. 证明该算法在最坏情形下为 $O(N)$ 。

b. 为什么该算法会优于课文中描述的算法？

6.26 合并图6-59中的两个斜堆。

6.27 写出将键1到15依序插入到一斜堆内的结果。

6.28 证明下述结论成立或不成立：如果将键1到 $2^k - 1$ 依序插入到一个初始为空的斜堆中，那么结果形成一棵理想平衡树。

6.29 使用标准的二叉堆算法可以建立一个 $N$ 个元素的斜堆。我们能否将练习6.25中描述的同样的合并策略用于斜堆而得到 $O(N)$ 运行时间？

6.30 证明二项树 $B_k$ 以二项树 $B_0, B_1, \dots, B_{k-1}$ 作为其根的儿子。

6.31 证明：高度为 $k$ 的二项树在深度 $d$ 有 $\binom{k}{d}$ 个结点。

6.32 将图6-60中的两个二项队列合并。

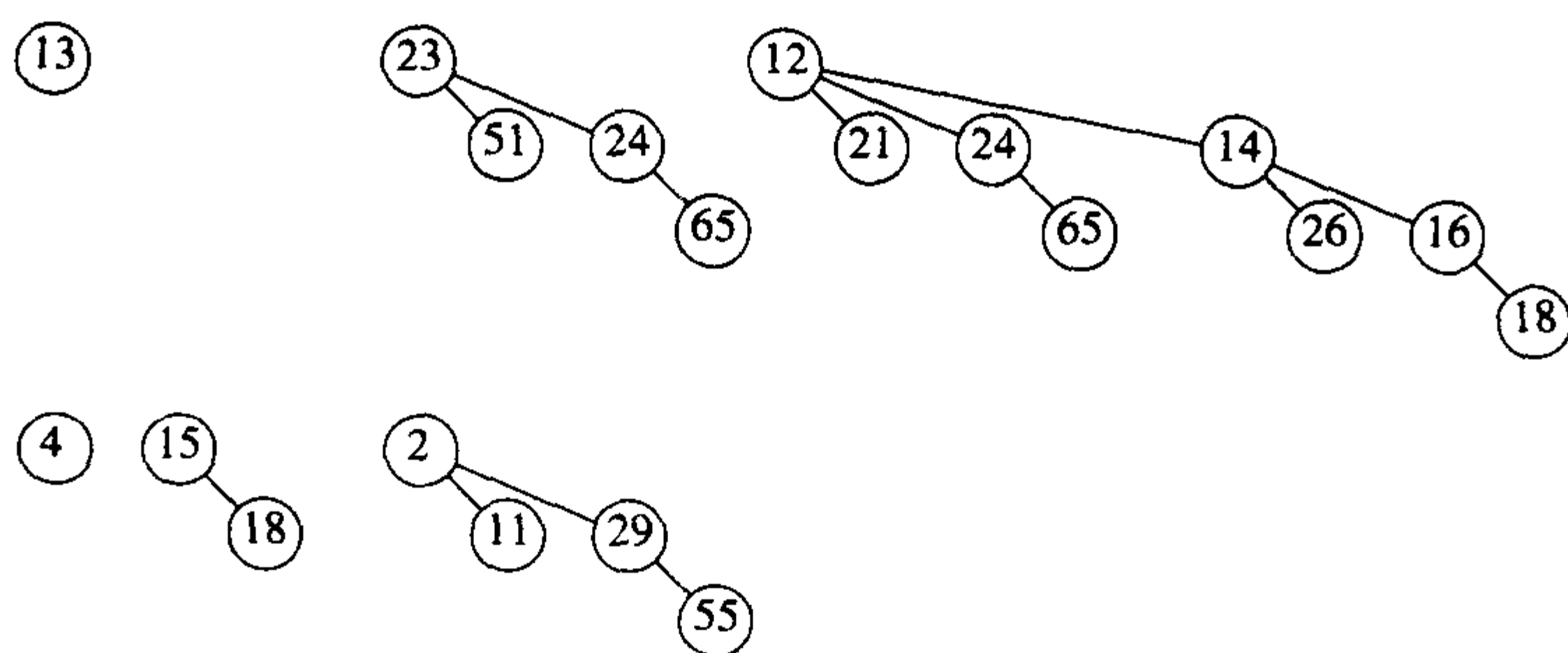


图6-60 练习6.32中的输入

6.33 a. 证明，向初始为空的二项队列进行 $N$ 次insert，在最坏情形下花费 $O(N)$ 的时间。

b. 给出一个算法来建立有 $N$ 个元素的二项队列，在元素间最多使用 $N-1$ 次比较。

\*c. 提出一个算法，以 $O(M + \log N)$ 最坏情形运行时间将 $M$ 个结点插入到 $N$ 个元素的二项队列中。证明该算法的时间界。

6.34 写出一个高效的例程，使用二项队列来完成insert操作。不要调用merge。

6.35 对于二项队列：

a. 如果没有树留在 $H_2$ 中且carry树为NULL，则修改merge例程以终止合并。

b. 修改merge使得较小的树总被合并到较大的树中。

\*\*6.36 假设我们将二项队列扩充为允许每个结构同一高度至多有两棵树。我们能否在保持其他操作为 $O(\log N)$ 时得到插入为 $O(1)$ 的最坏情形时间？

6.37 设有许多盒子，每个盒子都能容纳总重量 $C$ 和物品 $i_1, i_2, i_3, \dots, i_N$ ，它们分别重 $w_1, w_2, w_3, \dots, w_N$ 。现在想要把所有的物品包装起来，但任一盒子都不能放置超过其容量的重物，而且要使用尽量少的盒子。例如，若 $C = 5$ ，物品分别重2, 2, 3, 3，则我们可用两个盒子解决该问题。

一般说来, 这个问题很难, 尚不知有高效的解决方法。编写一个程序, 高效地实现下列各近似方法:

- \*a. 将重物放入能够承受其重量的第一个盒子内(如果没有盒子拥有足够的容量就开辟一个新的盒子)。(该方法以及后面所有的方法都将得出3个盒子, 这不是最优的结果。)
- b. 把重物放入对其有最大容量的盒子内。
- \*c. 把重物放入能够容纳下它而又不过载的装填得最满的盒子中。
- \*\*d. 这些方法有通过将重物按重量预先排序而功能得到增强的吗?

6.38 设我们想要将操作`decreaseAllKeys( $\Delta$ )`添加到堆的指令系统中去。该操作的结果是堆中所有的键都将它们的值减少量 $\Delta$ 。对于你所选择的堆的实现方法, 解释所做的必要修改使得所有其他操作都保持它们的运行时间而`decreaseAllKeys`以 $O(1)$ 运行。

6.39 两个选择算法中哪个具有更好的时间界?

6.40 对左式堆应用标准的`operator=`和`makeEmpty`可能会因为存在过多的递归调用而失败。虽然这些操作对二叉查找树是成立的, 但是对于左式堆还存在疑问, 因为虽然对基本的操作可能会有一个好的最坏情形时间界, 但左式堆可能会很深。因此, `operator=`和`makeEmpty`需要重新实现以避免在左式堆中过深的递归调用。按如下步骤完成这个新的实现:

- a. 记录递归调用例程, 这样对`t->left`的递归调用跟随对`t->right`的递归调用。
- b. 重写例程, 最后一句是对左子树的递归调用。
- c. 消除尾递归。
- d. 这些函数还是递归的。给出一个递归深度的精确时间界。
- \*e. 说明对于斜堆如何重写`operator=`和`makeEmpty`。

## 参考文献

二叉堆首先在[28]中描述。构造它的线性时间算法来自[14]。

*d*堆最初的描述见于[19]。最近的结果指出, 4叉堆在某些情形下可以改进二叉堆[22]。左式堆由Crane[11]发明并在Knuth[21]中描述。斜堆由Sleator和Tarjan[24]开发。二项队列由Vuillemin[27]发明; Brown提供了详细的分析和经验性的研究, 指出若能仔细地实现, 则它们在实践中性能很好[4]。

练习6.7(b-c)取自[17]。练习6.10(c)取自[6]。平均使用大约 $1.52N$ 次比较构造二叉堆的方法在[23]中描述。左式堆中的懒惰删除(练习6.24)来自[10]。练习6.36的一种解法可在[8]中查到。

最小—最大堆(练习6.18)的原始描述见于[1]。这些操作的更有效的实现在[18]和[25]中给出。双端优先队列(double-ended priority queues)的另外一些表示形式是*deap*和*diamond deque*。细节可见于[5]、[7]和[9]。练习6.18(e)的解法在[12]和[20]中给出。

理论上有趣的优先队列表示法是斐波那契堆(Fibonacci heap)[16], 我们将在第11章中描述它。斐波那契堆使得所有的操作都以 $O(1)$ 摊还时间执行, 删除操作除外, 它是 $O(\log N)$ 。松堆(relaxed heaps)[13]得到最坏情形下完全相同的界(merge操作除外)。[3]的过程对所有操作均得到最佳的最坏情形界。另外一种有趣的实现方法是配对堆(pairing heap)[15], 它在第12章描述。最后, 当数据由一些小的整数组成时仍能正常工作的优先队列在[2]和[26]中描述。

1. M. D. Atkinson, J. R. Sack, N. Santoro, and T. Strothotte, "Min-Max Heaps and Generalized Priority Queues," *Communications of the ACM*, 79 (1986), 996-1000.
2. J. D. Bright, "Range Restricted Mergeable Priority Queues," *Information Processing Letters*, 47 (1993), 159-164.
3. G. S. Brodal, "Worst-Case Efficient Priority Queues," *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (1996), 52-58.
4. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms," *SIAM Journal on Computing*, 7 (1978), 298-319.

5. S. Carlsson, "The Deap—A Double-Ended Heap to Implement Double-Ended Priority Queues," *Information Processing Letters*, 26 (1987), 33-36.
6. S. Carlsson and J. Chen, "The Complexity of Heaps," *Proceedings of the Third Symposium on Discrete Algorithms* (1992), 393-402.
7. S. Carlsson, J. Chen, and T. Strothotte, "A Note on the Construction of the Data Structure 'Deap'," *Information Processing Letters*, 31 (1989), 315-317.
8. S. Carlsson, J. I. Munro, and P. V. Poblete, "An Implicit Binomial Queue with Constant Insertion Time," *Proceedings of First Scandinavian Workshop on Algorithm Theory* (1988), 1-13.
9. S. C. Chang and M. W. Due, "Diamond Deque: A Simple Data Structure for Priority Deques," *Information Processing Letters*, 46 (1993), 231-237.
10. D. Cheriton and R. E. Tarjan, "Finding Minimum Spanning Trees," *SIAM Journal on Computing*, 5 (1976), 724-742.
11. C. A. Crane, "Linear Lists and Priority Queues as Balanced Binary Trees," *Technical Report STAN-CS-72-259*, Computer Science Department, Stanford University, Stanford, Calif., 1972.
12. Y. Ding and M. A. Weiss, "The Relaxed Min-Max Heap: A Mergeable Double-Ended Priority Queue," *Acta Informatica*, 30 (1993), 215-231.
13. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation," *Communications of the ACM*, 31 (1988), 1343-1354.
14. R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, 7 (1964), 701.
15. M. L. Fredman, R. Sedgwick, D. D. Sleator, and R. E. Tarjan, "The Pairing Heap: A New Form of Self-adjusting Heap," *Algorithmica*, 1 (1986), 111-129.
16. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM*, 34 (1987), 596-615.
17. G. H. Gonnet and J. I. Munro, "Heaps on Heaps," *SIAM Journal on Computing*, 15 (1986), 964-971.
18. A. Hasham and J. R. Sack, "Bounds for Min-max Heaps," *BIT*, 27 (1987), 315-323.
19. D. B. Johnson, "Priority Queues with Update and Finding Minimum Spanning Trees," *Information Processing Letters*, 4 (1975), 53-57.
20. C. M. Khoong and H. W. Leong, "Double-Ended Binomial Queues," *Proceedings of the Fourth Annual International Symposium on Algorithms and Computation* (1993), 128-137.
21. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., AddisonWesley, Reading, Mass., 1998.
22. A. LaMarca and R. E. Ladner, "The Influence of Caches on the Performance of Sorting," *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (1997), 370-379.
23. C.J.H. McDiarmid and B. A. Reed, "Building Heaps Fast," *Journal of Algorithms*, 10 (1989), 352-365.
24. D. D. Sleator and R. E. Tarjan, "Self-adjusting Heaps," *SIAM Journal on Computing*, 15 (1986), 52-69.
25. T. Strothotte, P. Eriksson, and S. Vallner, "A Note on Constructing Min-max Heaps," *BIT*, 29 (1989), 251-256.
26. P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and Implementation of an Efficient Priority Queue," *Mathematical Systems Theory*, 10 (1977), 99-127.
27. J. Vuillemin, "A Data Structure for Manipulating Priority Queues," *Communications of the ACM*, 21 (1978), 309-314.
28. J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, 7 (1964), 347-348.



# 排 序

**在**这一章，讨论将元素的数组排序的问题。为简单起见，假设在我们的例子中数组只包含整数，当然更复杂的对象也是可以的。对于本章的大部分内容，我们还假设整个排序工作能够在主存中完成，因此，元素的个数相对来说比较少（少于几百万）。当然，不能在主存中完成而必须在磁盘或磁带上完成的排序也相当重要。这种类型的排序叫作外部排序（external sorting），将在本章末尾进行讨论。

我们对内部排序的考查将指出：

- 存在几种容易的算法以 $O(N^2)$ 排序，如插入排序。
- 有一种算法叫作谢尔排序（Shellsort），它编程非常简单，以 $o(N^2)$ 运行，并在实践中很有效。
- 还有一些稍微复杂的 $O(M\log N)$ 的排序算法。
- 任何通用的排序算法均需要 $\Omega(M\log N)$ 次比较。

本章的其余部分将描述和分析各种排序算法。这些算法包含一些有趣的和重要的代码优化和算法设计思想。排序还是一种能够对其进行精确分析的算法的范例。预先声明，在适当的地方，我们将尽可能地多做一些分析。

## 7.1 预备知识

我们描述的算法都将是可互换的。每个算法都将接收包含一些元素的数组；假设所有的数组位置都包含要被排序的数据。我们还假设 $N$ 是传递到排序例程的元素的个数。

我们还假设存在“<”和“>”操作符，这些操作符可以用于将输入按一致的次序放置。除赋值运算外，这种运算是仅有的允许对输入数据进行的操作。在这些条件下的排序称为**基于比较的排序**（comparison-based sorting）。

这些接口与STL中的排序算法不同。在STL中，排序是通过使用函数模板sort来完成的。sort的参数反映了一个容器（的范围）的头尾标志，以及一个可选的比较器：

```
void sort( Iterator begin, Iterator end );
void sort( Iterator begin, Iterator end, Comparator cmp);
```

261

迭代器必须只是随机访问。sort算法不能保证相等的项保持它们原始的次序（如果这很重要的话，可以使用stable\_sort来替代sort）。

在下例中：

```
sort( v.begin(), v.end());
sort( v.begin(), v.end(), greater<int>());
sort( v.begin(), v.begin() + ( v.end() -v.begin())/2);
```

第一个调用将整个容器v按照非降序排列。第二个调用将整个容器按非升序排列。第三个调用将容器的前半部分按非降序排列。



在7.7节描述的排序算法通常用于快速排序。在7.2节，我们采用两种方法实现最简单的排序算法。一种是采用通过传递有可比较项的数组的方法，这可以得到最直观的代码。另一种是使用STL支持的接口，这需要更多的代码。

## 7.2 插入排序

### 7.2.1 算法

最简单的排序算法之一是插入排序（insertion sort）。插入排序由 $N-1$ 趟（pass）排序组成。对于 $p = 1$ 到 $N-1$ 趟，插入排序保证从位置0到位置 $p$ 上的元素为已排序状态。插入排序利用了这样的事实：位置0到位置 $p-1$ 上的元素是已经排过序的。图7-1显示了一个简单的数组在每一趟插入排序后的情况。

| 初始状态          | 34 | 8  | 64 | 51 | 32 | 21 | 移到的位置 |
|---------------|----|----|----|----|----|----|-------|
| After $p = 1$ | 8  | 34 | 64 | 51 | 32 | 21 | 1     |
| After $p = 2$ | 8  | 34 | 64 | 51 | 32 | 21 | 0     |
| After $p = 3$ | 8  | 34 | 51 | 64 | 32 | 21 | 1     |
| After $p = 4$ | 8  | 32 | 34 | 51 | 64 | 21 | 3     |
| After $p = 5$ | 8  | 21 | 32 | 34 | 51 | 64 | 4     |

图 7-1 每趟后的插入排序

图7-1表达了一般的策略。在第 $p$ 趟，我们将位置 $p$ 上的元素向左移动至它在前 $p+1$ 个元素中的正确位置上。图7-2中的程序实现了该策略。第11行到第14行实现数据移动而没有明显使用交换。位置 $p$ 上的元素存于tmp，而（在位置 $p$ 之前）所有更大的元素都被向右移动一个位置。然后将tmp置于正确的位置上。这种策略与实现二叉堆时所用到的技巧相同。

262

```
1  /**
2   * Simple insertion sort.
3   */
4  template <typename Comparable>
5  void insertionSort( vector<Comparable> & a )
6  {
7      int j;
8
9      for( int p = 1; p < a.size( ); p++ )
10     {
11         Comparable tmp = a[ p ];
12         for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
13             a[ j ] = a[ j - 1 ];
14         a[ j ] = tmp;
15     }
16 }
```

图7-2 插入排序例程

### 7.2.2 插入排序的STL实现

在STL中，排序例程不采用由具有可比性的项所组成的数组作为单一的参数，而是接收一对迭代器来代表在某范围内的起始和终止标志。一个双参数排序例程使用一对迭代器，并假设所有的项都可以排序。而一个三参数排序例程有一个函数对象作为第三个参数。

使用STL介绍的几个观点来转换图7-2中的算法。明显的观点是：

(1) 我们必须编写一个双参数排序和一个三参数排序的例程。假定双参数排序调用三参数排序，同时使用`less<Object>()`作为第三个参数。

(2) 数组访问必须转换成迭代器访问。

(3) 原始代码的11行需要创建`tmp`，在新代码中它将具有类型`Object`。

第一个观点最容易实现，因为模板类型参数（例如，泛型）对双参数排序来说都是`iterator`：然而，`Object`不是一个泛型参数。我们可以通过编写一个辅助例程来解决这个问题。如图7-3所示，这个辅助例程将`Object`作为另一个模板类型参数。

```

1  template <typename Iterator>
2  void insertionSort( const Iterator & begin, const Iterator & end )
3  {
4      if( begin != end )
5          insertionSortHelp( begin, end, *begin );
6  }
7
8
9  template <typename Iterator, typename Object>
10 void insertionSortHelp( const Iterator & begin, const Iterator & end,
11                        const Object & obj )
12 {
13     insertionSort( begin, end, less<Object>( ) );
14 }

```

图7-3 双参数排序通过使用一个辅助例程来调用三参数排序。该辅助例程将`Object`作为泛型类型处理

这里的小窍门是在双参数排序中，`*begin`具有类型`Object`，并且辅助例程具有所需要的第二个泛型的参数。现在双参数排序写完了，下面可以写三参数排序。但是，需要声明`tmp`为`Object`类型，三参数排序只具有`Iterator`和`Comparator`是泛型类型。因此我们不得不再次使用相同的技巧来得到一个四参数例程，将一个`Object`类型的项作为第四个参数，仅仅是为添加一个辅助的泛型类型。如图7-4所示，其中的代码使用迭代器来取代使用数组索引，使用`lessThan`函数对对象类取代了对`operator<`的调用。

263

```

1  template <typename Iterator, typename Comparator>
2  void insertionSort( const Iterator & begin, const Iterator & end,
3                    Comparator lessThan )
4  {
5      if( begin != end )
6          insertionSort( begin, end, lessThan, *begin );
7  }
8
9  template <typename Iterator, typename Comparator, typename Object>
10 void insertionSort( const Iterator & begin, const Iterator & end,
11                    Comparator lessThan, const Object & obj )
12 {
13     Iterator j;
14
15     for( Iterator p = begin+1; p != end; ++p )
16     {
17         Object tmp = *p;
18         for( j = p; j != begin && lessThan( tmp, *( j-1 ) ); --j )
19             *j = *(j-1);
20         *j = tmp;
21     }
22 }

```

图7-4 三参数排序调用四参数辅助例程，该辅助例程建立一个泛型类型的`Object`

观察我们以前编写的insertionSort算法的程序，原始程序中的每个语句都被一个新程序中的相应语句代替。新程序直接使用迭代器和函数对象。原始程序读起来很简单，这也是为什么我们在编写排序算法时使用更简单的界面而不是使用STL界面。

### 7.2.3 插入排序的分析

由于每一个嵌套循环都花费 $N$ 次迭代，因此插入排序为 $O(N^2)$ ，而且这个界是精确的，因为以反序的输入可以达到该界。精确计算指出，对于 $p$ 的每一个值，图7-2的测试最多执行 $p+1$ 次。对所有的 $p$ 求和，得到总数为：

$$\sum_{i=2}^N i = 2 + 3 + 4 + \dots + N = \Theta(N^2)$$

另一方面，如果输入数据已预先排序，那么运行时间为 $O(N)$ ，因为内层for循环的检测总是立即判定不成立而终止。事实上，如果输入几乎被排序（该术语将在下一节更严格地定义），那么插入排序将运行得很快。由于这种变化差别很大，因此值得我们去分析该算法平均情形的行为。实际上，和各种其他排序算法一样，插入排序的平均情形也是 $\Theta(N^2)$ ，详见下节的分析。

264

## 7.3 一些简单排序算法的下界

以数为成员的数组的逆序（inversion）是指具有性质 $i < j$  但 $a[i] > a[j]$ 的序偶 $(i, j)$ 。在上节的例子中，输入数据34, 8, 64, 51, 32, 21有9个逆序，即(34, 8)、(34, 32)、(34, 21)、(64, 51)、(64, 32)、(64, 21)、(51, 32)、(51, 21)以及(32, 21)。注意，这正好是需要由插入排序（隐含）执行的交换次数。情况总是这样，因为交换两个不按顺序排列的相邻元素恰好消除一个逆序，而一个排过序的数组没有逆序。由于算法中还有 $O(N)$ 项其他的工作，因此插入排序的运行时间是 $O(I+N)$ ，其中 $I$ 为原始数组中的逆序数。于是，若逆序数是 $O(N)$ ，则插入排序以线性时间运行。

可以通过计算排列中的平均逆序数而得出插入排序平均运行时间的精确的界。如往常一样，定义平均是很困难的命题。我们将假设不存在重复元素（如果允许重复，那么甚至连重复的平均次数究竟是什么都不清楚）。利用该假设，可设输入数据是前 $N$ 个整数的某个排列（因为只有相对顺序才是重要的），并设所有的排列都是等可能的。在这些假设下，我们有如下定理：

265

**定理7.1**  $N$ 个互异元素的数组的平均逆序数是 $N(N-1)/4$ 。

**证明** 对于任意的元素的表 $L$ ，考虑其反序表 $L_r$ 。上例中的反序表是21, 32, 51, 64, 8, 34。考虑该表中任意两个元素的序偶 $(x, y)$ ，且 $y > x$ 。显然，恰好是 $L$ 和 $L_r$ 中的一个，该序偶对应一个逆序。在表 $L$ 和它的反序表 $L_r$ 中，这样的序偶的总个数为 $N(N-1)/2$ 。因此，平均表有该量的一半，即 $N(N-1)/4$ 个逆序。 ■

这个定理意味着插入排序平均是二次的，同时也提供了只交换相邻元素的任何算法的一个很强的下界。

**定理7.2** 通过交换相邻元素进行排序的任何算法平均需要 $\Omega(N^2)$ 时间。

**证明** 初始的平均逆序数是 $N(N-1)/4 = \Omega(N^2)$ ，而每次交换只减少一个逆序，因此需要 $\Omega(N^2)$ 次交换。 ■

这是证明下界的一个例子，它不仅对隐含地实施相邻元素的交换的插入排序有效，而且对诸如冒泡排序和选择排序等其他一些简单算法也是有效的，不过我们在这里不讨论这些算法。事实上，它对所有的只进行相邻元素的交换的排序算法，包括那些未发现的算法，都是有效的。正因

为如此，这个证明在经验上是不能被认可的。虽然这个下界的证明非常简单，但是一般说来证明下界要比证明上界复杂得多。

这个下界告诉我们，为了使一个排序算法以亚二次（subquadratic）或 $o(N^2)$ 时间运行，必须执行一些比较，特别是要对相距较远的元素进行交换。排序算法通过删除逆序得以继续进行，而为了有效地进行，还必须每次交换删除多个逆序。

7.4 谢尔排序

谢尔排序（Shellsort）的名称源于它的发明者Donald Shell，该算法是冲破二次时间屏障的第一批算法之一，不过，直到它最初被发现的若干年后才证明了它的亚二次时间界。正如上节所提到的，它通过比较相距一定间隔的元素来工作；各趟比较所用的距离随着算法的进行而减小，直到只比较相邻元素的最后一趟排序为止。由于这个原因，谢尔排序有时也叫作**缩减增量排序**（diminishing increment sort）。

谢尔排序使用一个序列 $h_1, h_2, \dots, h_t$ ，叫作**增量序列**（increment sequence）。只要 $h_1 = 1$ ，任何增量序列都是可行的，不过，有些增量序列比另外一些增量序列更好（后面我们将讨论这个问题）。在使用增量 $h_k$ 的一趟排序之后，对于每一个 $i$ 我们有 $a[i] \leq a[i + h_k]$ （这里该不等式是有意义的）；所有相隔 $h_k$ 的元素都被排序。此时称文件是 **$h_k$ 排序的**（ $h_k$ -sorted）。例如，图7-5显示了几趟排序后数组的情况。谢尔排序的一个重要性质（我们只叙述而不证明）是，一个 $h_k$ 排序的文件（此后将是 $h_{k-1}$ 排序的）保持它的 $h_k$ 排序性。事实上，假如情况不是这样的话，那么该算法也就没什么意义了，因为前面各趟排序的成果会被后面各趟排序给打乱。

266

|        |    |    |    |    |    |    |    |    |    |    |    |    |    |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 初始状态   | 81 | 94 | 11 | 96 | 12 | 35 | 17 | 95 | 28 | 58 | 41 | 75 | 15 |
| 5 排序之后 | 35 | 17 | 11 | 28 | 12 | 41 | 75 | 15 | 96 | 58 | 81 | 94 | 95 |
| 3 排序之后 | 28 | 12 | 11 | 35 | 15 | 41 | 58 | 17 | 94 | 75 | 81 | 96 | 95 |
| 1 排序之后 | 11 | 12 | 15 | 17 | 28 | 35 | 41 | 58 | 75 | 81 | 94 | 95 | 96 |

图7-5 谢尔排序每趟之后的情况

$h_k$ 排序的一般做法是，对于 $h_k, h_k+1, \dots, N-1$ 中的每一个位置 $i$ ，把其上的元素放到 $i, i-h_k, i-2h_k, \dots$ 中间的正确位置上。虽然这并不影响最终结果，但是仔细的观察显示出，一趟 $h_k$ 排序的作用就是对 $h_k$ 个独立的子数组执行一次插入排序。当我们分析谢尔排序的运行时间时，这个观察结果将是很重要的。

增量序列的一个流行（但是不好）的选择是使用Shell建议的序列： $h_t = \lfloor N/2 \rfloor$ 和 $h_k = \lfloor h_{k+1}/2 \rfloor$ 。图7-6包含一个使用该序列实现谢尔排序的方法。后面我们将看到，存在一些递增的序列，它们对该算法的运行时间给出了重要的改进；即使是一个小的改变都可能剧烈地影响算法的性能（见练习7.10）。

267

图7-6中的程序以与插入排序实现方法中相同的方式避免显式地使用交换。

谢尔排序的最坏情形分析

虽然谢尔排序编程简单，但是，其运行时间的分析则完全是另外一回事。谢尔排序的运行时间依赖于增量序列的选择，而证明可能相当复杂。谢尔排序的平均情形分析，除最平凡的一些增量序列外，是一个长期未解决的问题。我们将证明在两个特别的增量序列下最坏情形的精确的界。



```

1  /**
2   * Shellsort, using Shell's (poor) increments.
3   */
4  template <typename Comparable>
5  void shellsort( vector<Comparable> & a )
6  {
7      for( int gap = a.size( ) / 2; gap > 0; gap /= 2 )
8          for( int i = gap; i < a.size( ); i++ )
9              {
10                 Comparable tmp = a[ i ];
11                 int j = i;
12
13                 for( ; j >= gap && tmp < a[ j - gap ]; j -= gap )
14                     a[ j ] = a[ j - gap ];
15                 a[ j ] = tmp;
16             }
17 }

```

图7-6 使用谢尔增量的谢尔排序例程（可能有更好的增量）

**定理7.3** 使用谢尔增量时谢尔排序的最坏情形运行时间为 $\Theta(N^2)$ 。

**证明** 证明不仅需要指出最坏情形运行时间的上界，而且还需要指出存在某个输入，实际上正好花费 $\Omega(N^2)$ 时间运行。首先通过构造一个不好的情形来证明下界。我们首先选择 $N$ 是2的幂，这使得除最后一个增量是1外所有的增量都是偶数。现在，给出一个数组作为输入，它的偶数位置上有 $N/2$ 个同是最大的数，而在奇数位置上有 $N/2$ 个同为最小的数（对该证明，第一个位置是位置1）。由于除最后一个增量外所有的增量都是偶数，因此，当我们进行最后一趟排序前， $N/2$ 个最大的元素仍然在偶数位置上，而 $N/2$ 个最小的元素也还是在奇数位置上。于是，在最后一趟排序开始之前第 $i$ 个最小的数（ $i \leq N/2$ ）在位置 $2i-1$ 上。将第 $i$ 个元素恢复到其正确位置需要在数组中移动 $i-1$ 个间隔。这样，仅仅将 $N/2$ 个最小的元素放到正确的位置上就需要至少 $\sum_{i=1}^{N/2} i-1 = \Omega(N^2)$ 的工作。作为一个例子，图7-7显示一个 $N=16$ 时的坏（但不是最坏）的输入。在2排序后的逆序数一直保持恰好为 $1+2+3+4+5+6+7=28$ ；因此，最后一趟排序将花费相当多的时间。

| 开始    | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6  | 14 | 7  | 15 | 8  | 16 |
|-------|---|---|---|----|---|----|---|----|---|----|----|----|----|----|----|----|
| 8 排序后 | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6  | 14 | 7  | 15 | 8  | 16 |
| 4 排序后 | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6  | 14 | 7  | 15 | 8  | 16 |
| 2 排序后 | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6  | 14 | 7  | 15 | 8  | 16 |
| 1 排序后 | 1 | 2 | 3 | 4  | 5 | 6  | 7 | 8  | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

图7-7 具有谢尔增量的谢尔排序的不好情形（位置编号从1到16）

现在我们显示上界 $O(N^2)$ 来结束本证明。正如前面我们已经看到的，带有增量 $h_k$ 的一趟排序由 $h_k$ 个关于 $N/h_k$ 个元素的插入排序组成。由于插入排序是二次的，因此一趟排序总的开销是 $O(h_k(N/h_k)^2) = O(N^2/h_k)$ 。对所有各趟排序求和则给出总的界为 $O(\sum_{i=1}^t N^2/h_i) = O(N^2 \sum_{i=1}^t 1/h_i)$ 。因为这些增量形成一个几何级数，其公比为2，而该级数中的最大项是 $h_1=1$ ，因此， $\sum_{i=1}^t 1/h_i < 2$ 。

268 于是，我们得到总的界 $O(N^2)$ 。 ■

谢尔增量的问题在于，这些增量对未必互素，因此较小的增量可能影响很小。Hibbard提出一个稍微不同的增量序列，它在实践中（与理论上）给出更好的结果。它的增量形如1, 3, 7, …，

$2^k-1$ 。虽然这些增量几乎是相同的，但关键的区别是相邻的增量没有公因子。现在我们就来分析使用这个增量序列的谢尔排序的最坏情形运行时间，这个证明相当复杂。

**定理7.4** 使用Hibbard增量的谢尔排序的最坏情形运行时间为  $\Theta(N^{3/2})$ 。

**证明** 我们只证明上界而将下界的证明留作练习。这个证明需要堆垒数论 (additive number theory) 中某些众所周知的结果。本章末提供了这些结果的参考资料。

和前面一样，对于上界，还是计算每一趟排序的运行时间的界，然后对各趟求和。对于那些  $h_k > N^{1/2}$  的增量，我们将使用前一定理得到的界  $O(N^2/h_k)$ 。虽然这个界对于其他增量也是成立的，但是它太大，用不上。直观地看，我们必须利用这个增量序列是特殊的这样一个事实。所需要证明的是，对于位置  $p$  上的任意元素  $a[p]$ ，当要执行  $h_k$  排序时，只有少数元素在位置  $p$  的左边且大于  $a[p]$ 。

当对输入数组进行  $h_k$  排序时，我们知道它已经是  $h_{k+1}$  排序和  $h_{k+2}$  排序的了。在  $h_k$  排序以前，考虑位置  $p$  和  $p-i$  上的两个元素，其中  $i \leq p$ 。如果  $i$  是  $h_{k+1}$  或  $h_{k+2}$  的倍数，那么显然  $a[p-i] < a[p]$ 。不仅如此，如果  $i$  可以表示为  $h_{k+1}$  和  $h_{k+2}$  的线性组合（以非负整数的形式），那么也有  $a[p-i] < a[p]$ 。作为一个例子，当我们进行3排序时，文件已经是7排序和15排序的了。52可以表示为7和15的线性组合： $52 = 1 \times 7 + 3 \times 15$ 。因此， $a[100]$  不可能大于  $a[152]$ ，因为  $a[100] \leq a[107] \leq a[122] \leq a[137] \leq a[152]$ 。

现在， $h_{k+2} = 2h_{k+1} + 1$ ，因此  $h_{k+1}$  和  $h_{k+2}$  没有公因子。在这种情形下，可以证明，至少和  $(h_{k+1} - 1)(h_{k+2} - 1) = 8h_k^2 + 4h_k$  一样大的所有整数都可以表示为  $h_{k+1}$  和  $h_{k+2}$  的线性组合（见本章末尾的参考文献）。

这就告诉我们，最内层 for 循环体对于这些  $N-h_k$  位置上的每一个位置最多执行  $8h_k + 4 = O(h_k)$  次。于是我们得到每趟的界为  $O(Nh_k)$ 。

利用大约一半的增量满足  $h_k < \sqrt{N}$  的事实并假设  $t$  是偶数，那么总的运行时间为：

$$O\left(\sum_{k=1}^{t/2} Nh_k + \sum_{k=t/2+1}^t N^2/h_k\right) = O\left(N \sum_{k=1}^{t/2} h_k + N^2 \sum_{k=t/2+1}^t 1/h_k\right)$$

因为两个和都是几何级数，并且  $h_{t/2} = \Theta(\sqrt{N})$ ，所以上式简化为：

$$= O(Nh_{t/2}) + O\left(\frac{N^2}{h_{t/2}}\right) = O(N^{3/2})$$

■ 269

使用Hibbard增量的谢尔排序的平均情形运行时间基于模拟的结果被认为是  $O(N^{5/4})$ ，但是没有人能够证明该结果。Pratt已经证明， $\Theta(N^{3/2})$  的界适用于广泛的增量序列。

Sedgewick提出了几种增量序列，其最坏情形运行时间（也是可以达到的）为  $O(N^{4/3})$ 。对于这些增量序列的平均运行时间猜测为  $O(N^{7/6})$ 。经验研究指出，在实践中这些序列的运行要比Hibbard的好得多，其中最好的是序列  $\{1, 5, 19, 41, 109, \dots\}$ ，该序列中的项或者是  $9 \times 4^i - 9 \times 2^i + 1$ ，或者是  $4^i - 3 \times 2^i + 1$ 。通过将这些值放到一个数组中可以容易地实现该算法。虽然有可能存在某个增量序列使得能够对谢尔排序的运行时间做出重大改进，但是，这个增量序列在实践中还是最为人们称道的。

关于谢尔排序还有几个其他结果，它们需要数论和组合数学中一些复杂的定理而且主要是在理论上有用。谢尔排序是算法非常简单且又具有极其复杂的分析的一个好例子。

谢尔排序的性能在实践中是完全可以接受的，即使是对于数以万计的  $N$  仍是如此。编程的简单特点使得它成为对适度的大量输入数据经常选用的算法。

## 7.5 堆排序

正如第6章提到的，优先队列可以用于以 $O(M\log N)$ 时间进行排序。基于该思想的算法叫作堆排序（heapsort），它给出我们至今所见到的最佳的大 $O$ 运行时间。

回忆在第6章建立 $N$ 个元素的二叉堆的基本方法，这个阶段花费 $O(N)$ 时间。然后执行 $N$ 次deleteMin操作。按照顺序，最小的元素先离开堆。通过这些元素记录到第二个数组然后再将数组拷贝回来，我们得到 $N$ 个元素的排序。由于每个deleteMin花费 $O(\log N)$ 时间，因此总的运行时间是 $O(M\log N)$ 。

该算法的主要问题在于，它使用了一个附加的数组。因此，存储需求增加一倍。在某些实例中这可能是个问题。注意，将第二个数组复制回第一个数组的附加时间消耗只是 $O(N)$ ，这不可能显著影响运行时间。这个问题是空间的问题。

避免使用第二个数组的聪明的方法是利用这样的事实：在每次deleteMin之后，堆缩小了1。因此，堆中最后的单元可以用来存放刚刚删去的元素。例如，设我们有一个堆，它含有6个元素。第一次deleteMin产生 $a_1$ 。现在该堆只有5个元素，因此可以把 $a_1$ 放在位置6上。下一次deleteMin产生 $a_2$ ，由于该堆现在只有4个元素，因此把 $a_2$ 放在位置5上。

使用这种策略，在最后一次deleteMin后，该数组将以递减顺序包含这些元素。如果想将这些元素排成更典型的递增顺序，那么可以改变堆序性质使得父亲的元素的值大于儿子的值。这样就得到（max）堆。

270

在我们的实现方法中将使用一个（max）堆，但由于速度的原因避免了具体的ADT。照通常的习惯，每一件事都是在数组中完成的。第一步以线性时间建立堆。然后通过将堆中的最后元素与第一个元素交换，缩减堆的大小并进行下滤，来执行 $N-1$ 次deleteMax操作。当算法终止时，数组则以所排的顺序包含这些元素。例如，考虑输入序列31, 41, 59, 26, 53, 58, 97。最后得到的堆如图7-8所示。

图7-9显示了第一次deleteMax之后的堆。从图中可以看出，堆中的最后元素是31；堆数组中放置97的那一部分从技术上说已不再属于该堆。在此后的5次deleteMax操作之后，该堆实际上只有一个元素，而在堆数组中留下的元素显示出的将是排序的顺序。

271

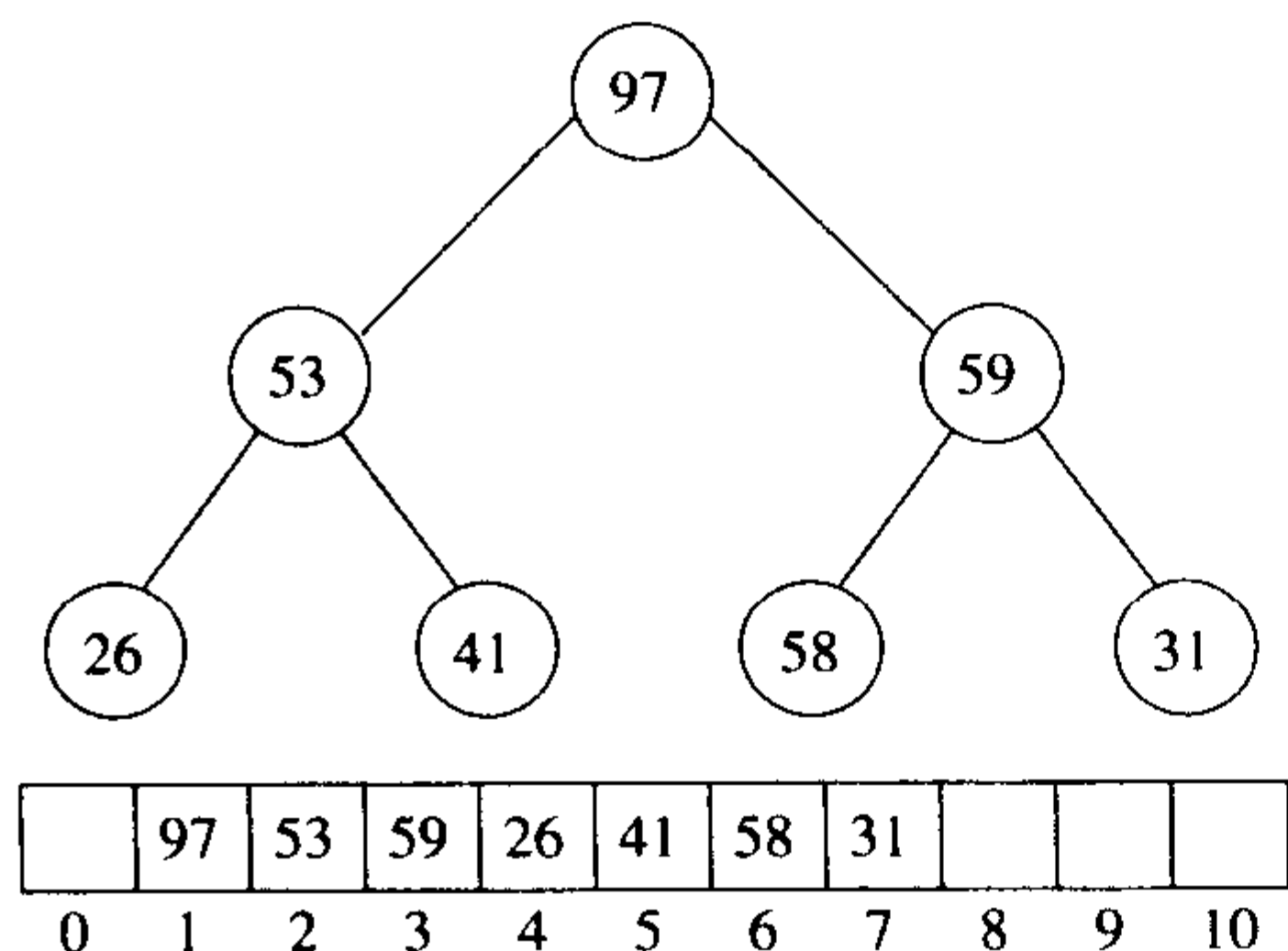


图7-8 在buildHeap阶段以后的(max)堆

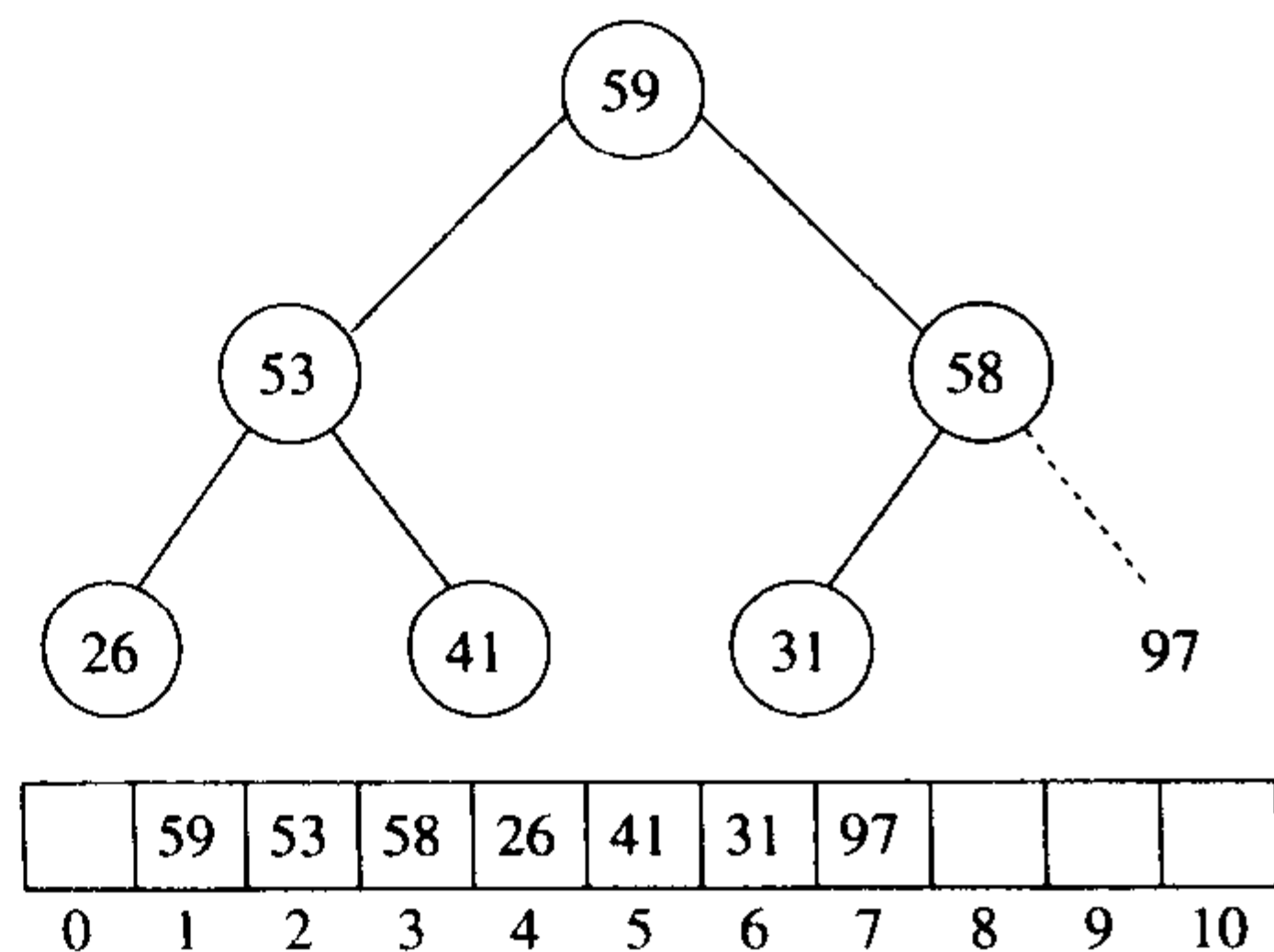


图7-9 在第一次deleteMax后的堆

执行堆排序的代码在图7-10中给出。稍微有些复杂的是，不像二叉堆，二叉堆的数据是从数组下标1处开始，而此处堆排序的数组包含位置0处的数据。因此，这里的程序与二叉堆的代码有些不同，不过变化很小。

```

1  /**
2   * Standard heapsort.
3   */
4  template <typename Comparable>
5  void heapsort( vector<Comparable> & a )
6  {
7      for( int i = a.size( ) / 2; i >= 0; i-- ) /* buildHeap */
8          percdDown( a, i, a.size( ) );
9      for( int j = a.size( ) - 1; j > 0; j-- )
10     {
11         swap( a[ 0 ], a[ j ] );          /* deleteMax */
12         percdDown( a, 0, j );
13     }
14 }
15
16 /**
17 * Internal method for heapsort.
18 * i is the index of an item in the heap.
19 * Returns the index of the left child.
20 */
21 inline int leftChild( int i )
22 {
23     return 2 * i + 1;
24 }
25
26 /**
27 * Internal method for heapsort that is used in deleteMax and buildHeap.
28 * i is the position from which to percolate down.
29 * n is the logical size of the binary heap.
30 */
31 template <typename Comparable>
32 void percdDown( vector<Comparable> & a, int i, int n )
33 {
34     int child;
35     Comparable tmp;
36
37     for( tmp = a[ i ]; leftChild( i ) < n; i = child )
38     {
39         child = leftChild( i );
40         if( child != n - 1 && a[ child ] < a[ child + 1 ] )
41             child++;
42         if( tmp < a[ child ] )
43             a[ i ] = a[ child ];
44         else
45             break;
46     }
47     a[ i ] = tmp;
48 }

```

图7-10 堆排序

## 堆排序的分析

我们在第6章看到，第一阶段构建堆最多用到 $2N$ 次比较。在第二阶段，第 $i$ 次deleteMax最多用到 $2\lfloor \log i \rfloor$ 次比较，总数最多为 $2\log N - O(N)$ 次比较（设 $N \geq 2$ ）。因此，在最坏的情形下堆排序最多使用 $2\log N - O(N)$ 次比较。练习7.13让你证明对于所有的deleteMax操作，有可能同时达到它们的最坏情形。

经验指出，堆排序是一个非常稳定的算法：它平均使用的比较只比最坏情形界指出的略少。



然而直到现在，还没有人能够指出堆排序平均运行时间的非平凡界。似乎问题在于连续的deleteMax操作破坏了堆的随机性，使得概率论证非常复杂。最近，另一种处理方法被证明是成功的。

**定理7.5** 对 $N$ 个互异项的随机排列进行堆排序，所用的比较平均次数为 $2M\log N - O(M\log \log N)$ 。

**证明** 构建堆的阶段平均使用 $\Theta(N)$ 次比较，因此我们只需要证明第二阶段的界。设有 $\{1, 2, \dots, N\}$ 的一个排列。

设第 $i$ 次deleteMax将根元素向下推了 $d_i$ 层。此时它使用了 $2d_i$ 次比较。对于任意的输入数据的堆排序，存在一个开销序列（cost sequence） $D: d_1, d_2, \dots, d_N$ ，它确定了第二阶段的开销，该开销由 $M_D = \sum_{i=1}^N d_i$ 给出；因此所使用的比较次数是 $2M_D$ 。

令 $f(N)$ 是 $N$ 项的堆的个数。可以证明（练习7.53） $f(N) > (N/(4e))^N$ （其中， $e = 2.71828\dots$ ）。我们将证明，只有这些堆中指数上很小的部分（特别是 $(N/16)^N$ ）的开销小于 $M = N(\log N - \log \log N - 4)$ 。当该结论得证时可以推出， $M_D$ 的平均值至少是 $M$ 减去大小为 $o(1)$ 的一项，这样，比较的平均次数至少是 $2M$ 。因此，我们的基本目标则是证明存在很少的具有小开销序列的堆。

因为第 $d_i$ 层上最多有 $2^{d_i}$ 个结点，所以对于任意的 $d_i$ ，存在根元素可能到达的 $2^{d_i}$ 个可能的位置。于是，对任意的序列 $D$ ，对应deleteMax的互异序列的个数最多是：

$$S_D = 2^{d_1} 2^{d_2} \dots 2^{d_N}$$

简单的代数处理指出，对一个给定的序列 $D$ ：

$$S_D = 2^{M_D}$$

272

因为每个 $d_i$ 可取1和 $\lfloor \log N \rfloor$ 之间的任一值，所以最多存在 $(\log N)^N$ 个可能的序列 $D$ 。由此可知，需要花费的开销恰好为 $M$ 的互异deleteMax序列的个数，最多是总开销为 $M$ 的开销序列的个数乘以每个这种开销序列的deleteMax序列的个数。这样就立刻得到界 $(\log N)^N 2^M$ 。

开销序列小于 $M$ 的堆的总数最多为：

$$\sum_{i=1}^{M-1} (\log N)^N 2^i < (\log N)^N 2^M$$

如果我们选择 $M = N(\log N - \log \log N - 4)$ ，那么开销序列小于 $M$ 的堆的个数最多为 $(N/16)^N$ ，根据前面的评述，定理得证。 ■

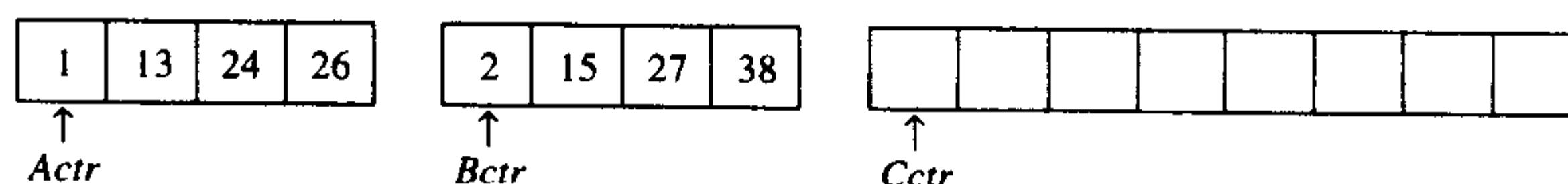
通过更复杂的论述，可以证明，堆排序总是使用至少 $M\log N - O(N)$ 次比较，而且存在能够达到这个界的输入数据。似乎平均情形也应该是 $2M\log N - O(N)$ 次比较（而不是定理7.5中非线性的第二项）；这是否能够证明（甚至是否成立）还是个尚未解决的问题。

## 7.6 归并排序

下面讨论归并排序（mergesort）。归并排序以 $O(M\log N)$ 最坏情形运行时间运行，而所使用的比较次数几乎是最优的。它是递归算法的一个很好的实例。

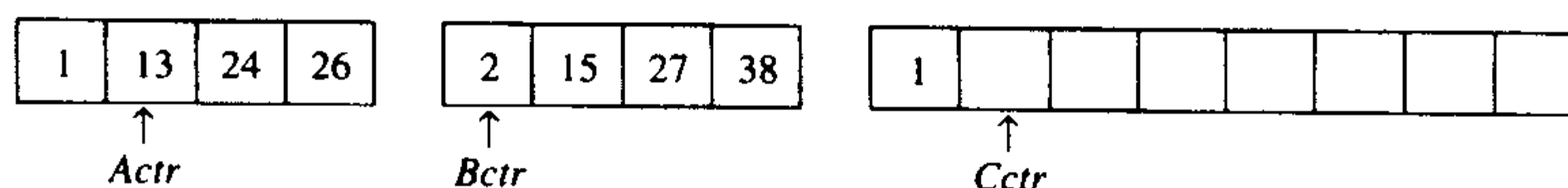
这个算法中基本的操作是合并两个已排序的表。因为这两个表是已排序的，所以若将输出放到第3个表中则该算法可以通过对输入数据一趟排序来完成。基本的合并算法是取两个输入数组 $A$ 和 $B$ 、一个输出数组 $C$ 以及3个计数器（ $Actr$ 、 $Bctr$ 和 $Cctr$ ），它们初始置于对应数组的开始端。 $A[Actr]$ 和 $B[Bctr]$ 中的较小者被复制到 $C$ 中的下一个位置，相关的计数器向前推进一步。当两个输入表

有一个用完的时候，则将另一个表中的剩余部分拷贝到C中。合并例程如何工作的例子见下。

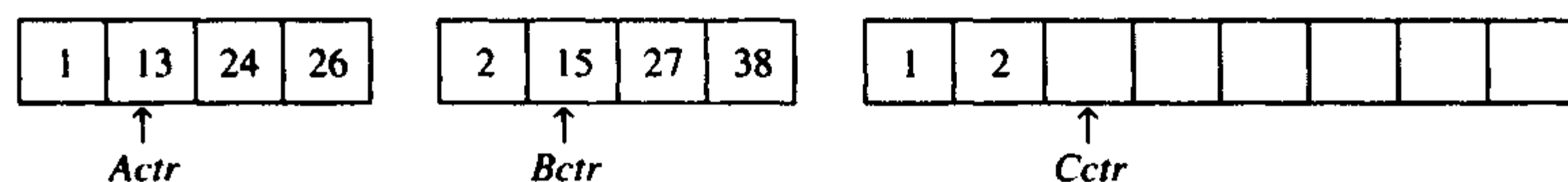


如果数组A含有1、13、24、26，数组B含有2、15、27、38，那么该算法如下进行：首先，比较在1和2之间进行，1被添加到C中，然后13和2进行比较。

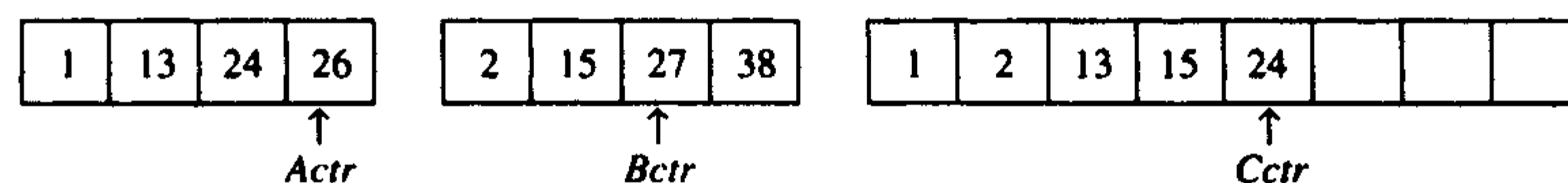
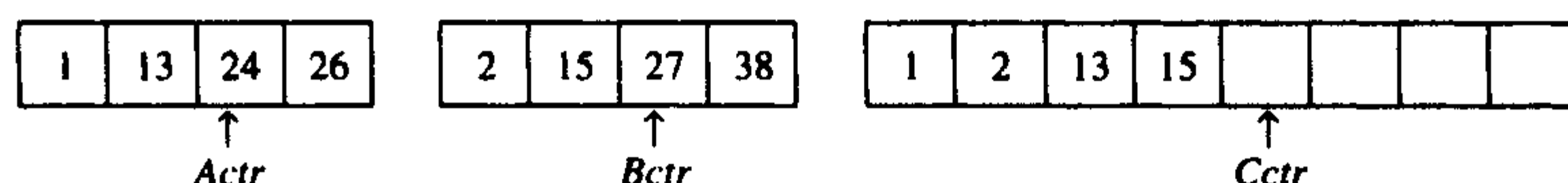
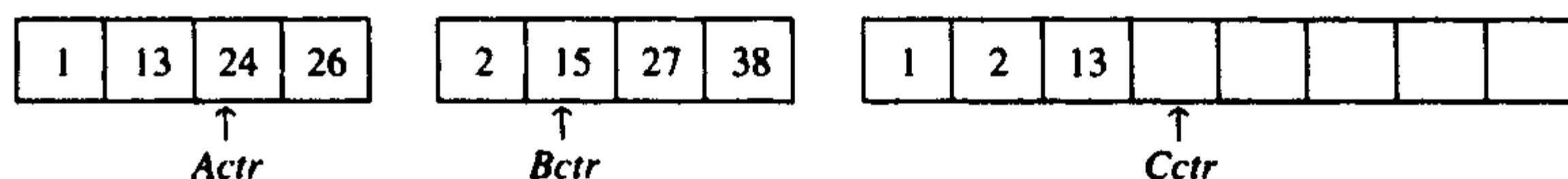
273  
274



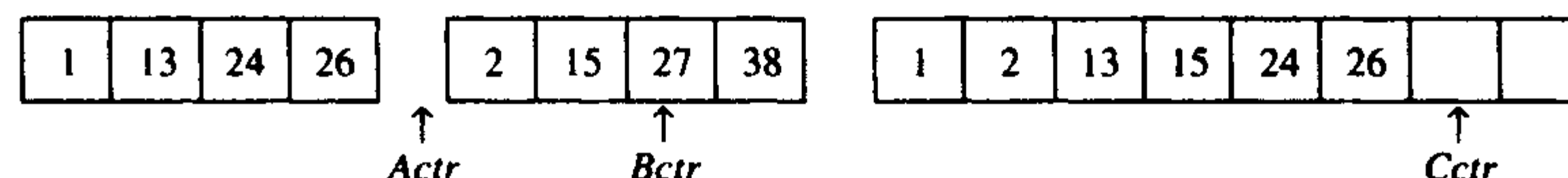
2被添加到C中，然后13和15进行比较。



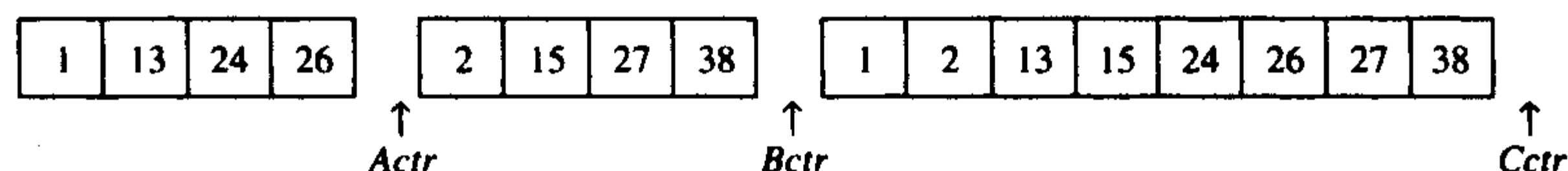
13被添加到C中，接下来比较24和15，这样一直到26和27进行比较。



26被添加到C中，数组A已经用完。



将数组B的其余部分复制到C中。



合并两个已排序的表的时间显然是线性的，因为最多进行了 $N-1$ 次比较，其中 $N$ 是元素的总数。为了看清这一点，注意每次比较都是把一个元素加到C中，但最后一次比较除外，它至少添加两个元素。

因此，归并排序算法很容易描述。如果 $N = 1$ ，那么只有一个元素需要排序，答案是显而易见的。否则，递归地将前半部分数据和后半部分数据各自归并排序，得到排序后的两部分数据，然后再使用上面描述的合并算法将这两部分合并到一起。例如，欲将8元素数组24, 13, 26, 1, 2, 27, 38, 15排序，我们递归地将前4个数据和后4个数据分别排序，得到1, 13, 24, 26, 2, 15, 27, 38。然后，将这两部分合并，得到最后的表1, 2, 13, 15, 24, 26, 27, 38。该算法是经典的分治(divide-and-conquer)策略，它将问题分(divide)成一些小的问题然后递归求解，而治(conquering)的阶段则是将分的阶段解得的各答案修补在一起。分治是递归非常有力的用法，我们将会多次

遇到。

归并排序的一种实现在图7-11中给出。单参数mergeSort是四参数递归函数mergeSort的一个驱动程序。

275

```

1  /**
2  * Mergesort algorithm (driver).
3  */
4  template <typename Comparable>
5  void mergeSort( vector<Comparable> & a )
6  {
7      vector<Comparable> tmpArray( a.size( ) );
8
9      mergeSort( a, tmpArray, 0, a.size( ) - 1 );
10 }
11
12 /**
13 * Internal method that makes recursive calls.
14 * a is an array of Comparable items.
15 * tmpArray is an array to place the merged result.
16 * left is the left-most index of the subarray.
17 * right is the right-most index of the subarray.
18 */
19 template <typename Comparable>
20 void mergeSort( vector<Comparable> & a,
21                vector<Comparable> & tmpArray, int left, int right )
22 {
23     if( left < right )
24     {
25         int center = ( left + right ) / 2;
26         mergeSort( a, tmpArray, left, center );
27         mergeSort( a, tmpArray, center + 1, right );
28         merge( a, tmpArray, left, center + 1, right );
29     }
30 }

```

图7-11 归并排序例程

merge例程是精妙的。如果对merge的每个递归调用均局部声明一个临时数组，那么在任一时刻就可能有 $\log N$ 个临时数组处在活动期。严密的考察指出，由于merge是mergeSort的最后一行，因此在任一时刻只需要一个临时数组活动，而且这个临时数组可以在mergeSort驱动程序中建立。不仅如此，还可以使用临时数组的任意部分；我们将使用与输入数组a相同的部分，这就达到本节末尾描述的改进。图7-12实现了这个merge例程。

## 归并排序的分析

276

归并排序是用于分析递归例程技巧的经典实例：必须给运行时间写出一个递推关系。假设 $N$ 是2的幂，从而总可以将它分裂成相等的两部分。对于 $N=1$ ，归并排序所用的时间是常数，我们将其记为1。否则，对 $N$ 个数归并排序的用时等于完成两个大小为 $N/2$ 的递归排序所用的时间再加上合并的时间，它是线性的。下述方程给出准确的表示：

$$T(1) = 1$$

$$T(N) = 2T(N/2) + N$$

277

这是一个标准的递推关系，它可以用多种方法求解。我们将介绍两种方法。第一种方法是用 $N$ 去除递推关系的两边（稍后将看出这么做的理由），相除后得到

```

1  /**
2   * Internal method that merges two sorted halves of a subarray.
3   * a is an array of Comparable items.
4   * tmpArray is an array to place the merged result.
5   * leftPos is the left-most index of the subarray.
6   * rightPos is the index of the start of the second half.
7   * rightEnd is the right-most index of the subarray.
8   */
9  template <typename Comparable>
10 void merge( vector<Comparable> & a, vector<Comparable> & tmpArray,
11             int leftPos, int rightPos, int rightEnd )
12 {
13     int leftEnd = rightPos - 1;
14     int tmpPos = leftPos;
15     int numElements = rightEnd - leftPos + 1;
16
17     // Main loop
18     while( leftPos <= leftEnd && rightPos <= rightEnd )
19         if( a[ leftPos ] <= a[ rightPos ] )
20             tmpArray[ tmpPos++ ] = a[ leftPos++ ];
21         else
22             tmpArray[ tmpPos++ ] = a[ rightPos++ ];
23
24     while( leftPos <= leftEnd )    // Copy rest of first half
25         tmpArray[ tmpPos++ ] = a[ leftPos++ ];
26
27     while( rightPos <= rightEnd ) // Copy rest of right half
28         tmpArray[ tmpPos++ ] = a[ rightPos++ ];
29
30     // Copy tmpArray back
31     for( int i = 0; i < numElements; i++, rightEnd-- )
32         a[ rightEnd ] = tmpArray[ rightEnd ];
33 }

```

图7-12 merge例程

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

该方程对2的幂的任意N是成立的，于是还可以写成

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

和

$$\begin{aligned} \frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + 1 \\ &\vdots \\ \frac{T(2)}{2} &= \frac{T(1)}{1} + 1 \end{aligned}$$

将所有这些方程相加，也就是说，将等号左边的所有各项相加并使结果等于右边所有各项的和。项 $T(N/2)/(N/2)$ 出现在等号两边可以消去。事实上，实际出现在两边的项均被消去，我们称之为叠缩（telescoping）求和。在所有的加法完成之后，最后的结果为

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

这是因为所有其余的项都被消去了而方程的个数是 $\log N$ ，故而将各方程末尾的1相加起来得到 $\log N$ 。再将两边同乘以N，得到最后的答案



$$T(N) = M \log N + N = O(M \log N)$$

注意，假如我们在求解开始时不是通除以 $N$ ，那么两边的和也就不可能叠缩。这就是等号两边要同时除以 $N$ 的原因。

另一种方法是在右边连续地代入递推关系，得到

$$T(N) = 2T(N/2) + N$$

由于可以将 $N/2$ 代入到主要的方程中，

$$2T(N/2) = 2(2(T(N/4)) + N/2) = 4T(N/4) + N$$

因此得到

$$T(N) = 4T(N/4) + 2N$$

再将 $N/4$ 代入到主要的方程中，我们看到

$$4T(N/4) = 4(2T(N/8) + N/4) = 8T(N/8) + N$$

因此有

$$T(N) = 8T(N/8) + 3N$$

按这种方式继续下去，得到

$$T(N) = 2^k T(N/2^k) + k \cdot N$$

利用 $k = \log N$ ，得到

$$T(N) = NT(1) + M \log N = M \log N + N$$

选择哪种方法是个人习惯问题。第一种方法引起一些琐碎的工作，把它写到一张 $8\frac{1}{2} \times 11$ 的纸上可能更好，这样会少出些数学错误，不过需要有一定的经验。第二种方法更偏重于使用蛮力进行计算。

该例子中，假设 $N = 2^k$ 。分析可以精化，以处理 $N$ 不是2的幂的情形。事实上，答案几乎是一样的（通常出现的就是这样的情形）。

虽然归并排序的运行时间是 $O(M \log N)$ ，但是它很难用于主存排序，主要问题在于合并两个排序的表需要线性附加内存，在整个算法中还要花费将数据复制到临时数组再复制回来这样一些附加的工作，其结果是严重减慢了排序的速度。这种复制可以通过在递归的交替层面上审慎地交换 $a$ 和 $tmpArray$ 的角色加以避免。归并排序的一种变形也可以非递归地实现（见练习7.16）。

与其他的 $O(M \log N)$ 排序相比，归并排序的运行时间很大程度上依赖于在数组中进行元素的比较和移动所消耗的时间。这些消耗是和编程语言相关的。

例如，在其他语言（例如Java）中，当排序一般的对象时，元素的比较耗时很多，但是移动元素就快得多。在所有流行的排序算法中，归并排序使用最少次数的比较。因此，在Java中，归并排序是一般目的排序的最佳选择。事实上，在标准Java库中的一般排序就是用的这种算法。

另一方面，在C++中，对于一般排序，当对象很大时，复制对象的代价是很大的，而对象的比较通常相对消耗小些。这是因为编译器在处理函数模板的扩展时具有强大的执行在线优化的能力。在本节中，如果我们可以使用很少的数据移动，那么即使使用稍微多一些比较的算法也是合理的。下一节描述的快速排序算法较好地平衡了这两者，而且也是C++库中普遍使用的排序例程。

## 7.7 快速排序

顾名思义，快速排序（quicksort）是在实践中最快的已知排序算法，它的平均运行时间是 $O(M \log N)$ 。该算法之所以特别快，主要是由于非常精炼和高度优化的内部循环。它的最坏情形的

性能为 $O(N^2)$ ，但稍加努力就可避免这种情形。通过将堆排序与快速排序结合起来，就可以在堆排序的 $O(M\log N)$ 最坏运行时间下，得到对几乎所有输入的最快运行时间。练习7.27描述了这一方法。

虽然多年来快速排序算法曾被认为是理论上高度优化而在实践中不可能正确编程的一种算法，但是该算法简单易懂而且不难证明。像归并排序一样，快速排序也是一种分治的递归算法。将数组 $S$ 排序的基本算法由下列简单的四步组成：

- (1) 如果 $S$ 中元素个数是0或1，则返回。
- (2) 取 $S$ 中任一元素 $v$ ，称之为枢纽元 (pivot)。
- (3) 将 $S-\{v\}$  ( $S$ 中其余元素) 划分成两个不相交的集合： $S_1 = \{x \in S-\{v\} \mid x \leq v\}$  和  $S_2 = \{x \in S-\{v\} \mid x \geq v\}$ 。
- (4) 返回 $\{\text{quicksort}(S_1), \text{后跟}v, \text{继而}\text{quicksort}(S_2)\}$ 。

由于在那些等于枢纽元的元素的处理上，第3步划分的描述不是唯一的，因此这就成了设计决策。一部分好的实现方法是将这种情形尽可能有效地处理。直观地看，我们希望把等于枢纽元的大约一半的键分到 $S_1$ 中，而另外的一半分到 $S_2$ 中，很像我们希望二叉查找树保持平衡的情形。

图7-13解释了如何快速排序一个数集。这里的枢纽元（随机地）选为65，集合中其余元素分成两个更小的集合。递归地将较小的数的集合排序得到0, 13, 26, 31, 43, 57（递归法则3），将较大的数的集合类似排序，此时很容易得到整个集合的排序。

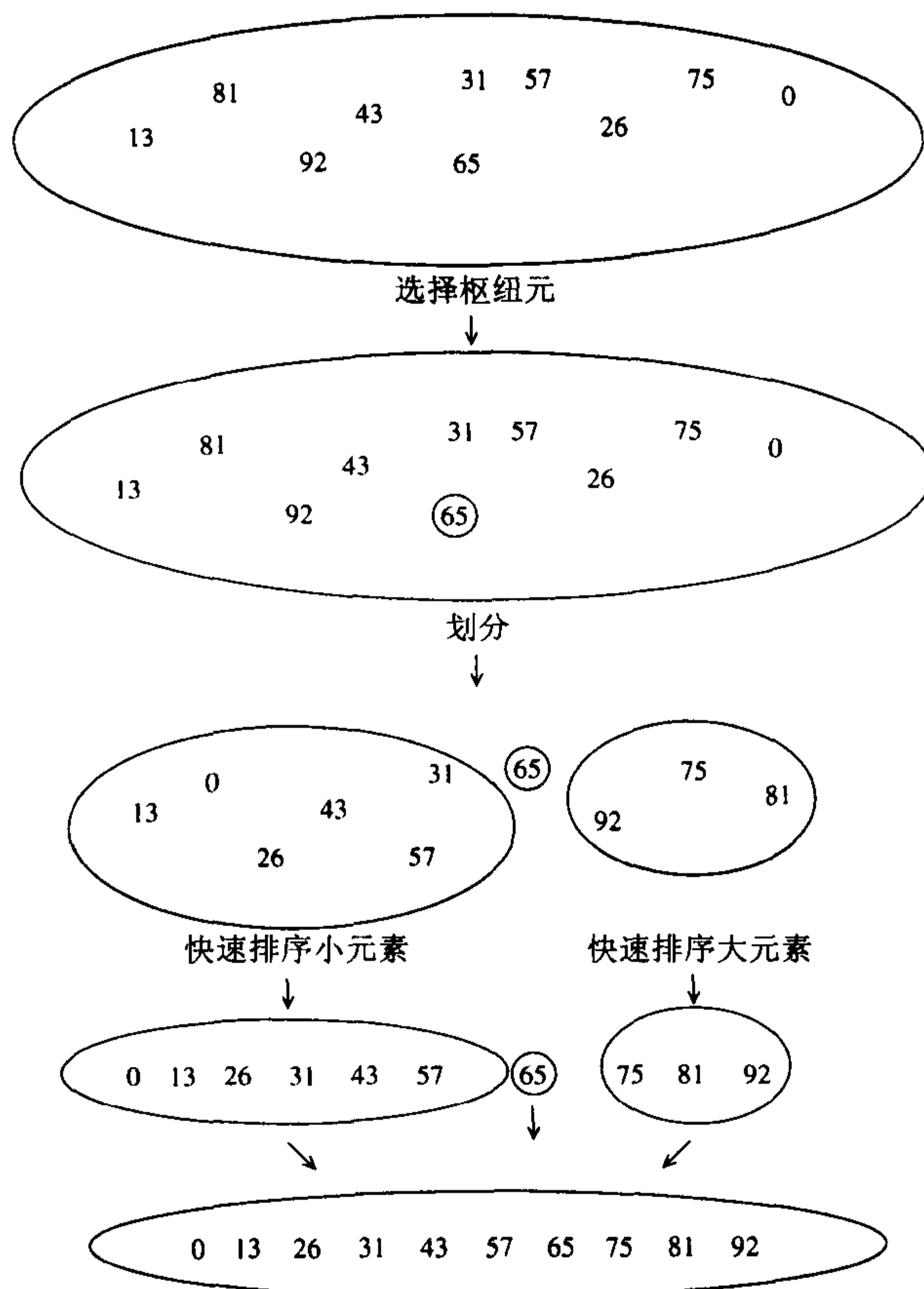


图7-13 说明快速排序各步的演示示例

显然该算法成立，但是不清楚的是，为什么它比归并排序快。如同归并排序那样，快速排序递归地解决两个子问题并需要线性的附加工作（第3步），不过，与归并排序不同，这两个子问题并不保证具有相等的大小，这是个潜在的隐患。快速排序更快的原因在于，第3步划分成两组实际上是在适当的位置进行并且非常有效，它的高效不仅弥补了大小不等的递归调用的不足而且还超过了它。

迄今为止对该算法的描述尚缺少许多细节，我们现在就来补充这些细节。实现第2步和第3步有许多方法；这里介绍的方法是大量分析和经验研究的结果，它代表实现快速排序的非常有效的方法，哪怕是对该方法最微小的偏差都可能引起意想不到的不良结果。

### 7.7.1 选取枢纽元

虽然上面描述的算法无论选择哪个元素作为枢纽元都能完成排序工作，但是有些选择显然更优。

#### 1. 一种错误的方法

通常的、没有经过充分考虑的选择是将第一个元素用作枢纽元。如果输入是随机的，那么这是可以接受的，但是如果输入是预排序的或是反序的，那么这样的枢纽元就产生一个劣质的分割，因为所有的元素不是都被划入 $S_1$ ，就是都被划入 $S_2$ 。更有甚者，这种情况可能发生在所有的递归调用中。实际上，如果第一个元素用作枢纽元而且输入是预先排序的，那么快速排序花费的时间将是二次的，可是实际上却根本没做什么事，这是相当尴尬的。然而，预排序的输入（或具有一大段预排序数据的输入）是相当常见的，因此，使用第一个元素作为枢纽元是非常糟糕的，应该立即放弃这种想法。另一种想法是选取前两个互异的键中的较大者作为枢纽元，但这和只选取第一个元素作为枢纽元具有相同的害处。不要使用这两种选取枢纽元的策略。

[280]

#### 2. 一种安全的做法

一种安全的方针是随机选取枢纽元。一般来说这种策略非常安全，除非随机数生成器有问题（这不像你所想像的那么罕见），因为随机的枢纽元不可能总在接连不断地产生劣质的分割。另一方面，随机数的生成一般是昂贵的，根本减少不了算法其余部分的平均运行时间。

[281]

#### 3. 三数中值分割法

一组 $N$ 个数的中值是第 $\lceil N/2 \rceil$ 个最大的数。枢纽元的最好的选择是数组的中值。可是，这很难算出，并且会明显减慢快速排序的速度。这样的中值的估计可以通过随机选取三个元素并用它们的中值作为枢纽元而得到。事实上，随机性并没有多大的帮助，因此一般的做法是使用左端、右端和中心位置上的三个元素的中值作为枢纽元。例如，输入为8, 1, 4, 9, 6, 3, 5, 2, 7, 0，它的左边元素是8，右边元素是0，中心位置上的元素是6。于是枢纽元则是 $v = 6$ 。显然使用三数中值分割法消除了预排序输入的不好情形（在这种情形下，这些分割都是一样的），并且减少了快速排序大约14%的比较次数。

### 7.7.2 分割策略

有几种分割策略用于实践，但是已知此处描述的分割策略能够给出好的结果。我们将会看到，这么做很容易出错或产生低效率，但使用一种已知方法却是安全的。该方法的第一步是通过将枢纽元与最后的元素交换使得枢纽元离开要被分割的数据段。 $i$ 从第一个元素开始而 $j$ 从倒数第二个元素开始。如果最初的输入与前面一样，那么下面的图表示当前的状态。

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 1 | 4 | 9 | 0 | 3 | 5 | 2 | 7 | 6 |
| ↑ |   |   |   |   |   |   |   | ↑ |   |
| i |   |   |   |   |   |   |   | j |   |

我们暂时假设所有的元素互异，后面将着重考虑在出现重复元素时应该怎么办。作为一种限制性的情形，如果所有的元素都相同，那么我们的算法必须做相应的工作。然而奇怪的是，此时做错事却特别地容易。

在分割阶段要做的就是把所有小元素移到数组的左边而把所有大元素移到数组的右边。当然，“小”和“大”是相对于枢纽元而言的。

当 $i$ 在 $j$ 的左边时，我们将 $i$ 右移，移过那些小于枢纽元的元素，并将 $j$ 左移，移过那些大于枢纽元的元素。当 $i$ 和 $j$ 停止时， $i$ 指向一个大元素而 $j$ 指向一个小元素。如果 $i$ 在 $j$ 的左边，那么将这两个元素互换，其效果是把一个大元素推向右边而把一个小元素推向左边。在上面的例子中， $i$ 不移动，而 $j$ 滑过一个位置，情况如下图。

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 1 | 4 | 9 | 0 | 3 | 5 | 2 | 7 | 6 |
| ↑ |   |   |   |   |   |   | ↑ |   |   |
| i |   |   |   |   |   |   | j |   |   |

然后交换由 $i$ 和 $j$ 指向的元素，重复该过程直到 $i$ 和 $j$ 彼此交错为止。

282

第一次交换之后

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 4 | 9 | 0 | 3 | 5 | 8 | 7 | 6 |
| ↑ |   |   |   |   |   |   | ↑ |   |   |
| i |   |   |   |   |   |   | j |   |   |

第二次交换之前

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 4 | 9 | 0 | 3 | 5 | 8 | 7 | 6 |
|   |   |   | ↑ |   |   | ↑ |   |   |   |
|   |   |   | i |   |   | j |   |   |   |

第二次交换之后

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 4 | 5 | 0 | 3 | 9 | 8 | 7 | 6 |
|   |   |   | ↑ |   |   | ↑ |   |   |   |
|   |   |   | i |   |   | j |   |   |   |

第三次交换之前

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 4 | 5 | 0 | 3 | 9 | 8 | 7 | 6 |
|   |   |   |   |   | ↑ | ↑ |   |   |   |
|   |   |   |   |   | j | i |   |   |   |

此时， $i$ 和 $j$ 已经交错，故不再交换。分割的最后一步是将枢纽元与 $i$ 所指向的元素交换。

与枢纽元交换之后

|   |   |   |   |   |   |   |   |   |       |
|---|---|---|---|---|---|---|---|---|-------|
| 2 | 1 | 4 | 5 | 0 | 3 | 6 | 8 | 7 | 9     |
|   |   |   |   |   |   | ↑ |   |   | ↑     |
|   |   |   |   |   |   | i |   |   | pivot |

在最后一步，当枢纽元与 $i$ 所指向的元素交换时，我们知道在位置 $p < i$ 的每一个元素都必然是小元素，这是因为或者位置 $p$ 包含一个从它开始移动的小元素，或者位置 $p$ 上原来的大元素在交



换期间被置换了。类似的论断指出，在位置 $p > i$ 上的元素必然都是大元素。

我们必须考虑的一个重要的细节是如何处理那些等于枢纽元的元素。问题在于，当 $i$ 遇到一个等于枢纽元的元素时是否应该停止以及当 $j$ 遇到一个等于枢纽元的元素时是否应该停止。直观地看， $i$ 和 $j$ 应该做相同的工作，因为否则分割将出现偏向一方的倾向。例如，如果 $i$ 停止而 $j$ 不停，那么所有等于枢纽元的元素都将被分到 $S_2$ 中。

283

为了搞清怎么办更好，我们考虑数组中所有的元素都相等的情况。如果 $i$ 和 $j$ 都停止，那么在相等的元素间将有很多次交换。虽然这似乎没有什么意义，但是其正面的效果则是 $i$ 和 $j$ 将在中间交错，因此当枢纽元被替代时，这种分割建立了两个几乎相等的子数组。归并排序的分析告诉我们，此时总的运行时间为 $O(M \log N)$ 。

如果 $i$ 和 $j$ 都不停止，那么就应该有相应的程序防止 $i$ 和 $j$ 越出数组的端点，不进行交换的操作。虽然这样似乎不错，但是正确的实现方法却是把枢纽元交换到 $i$ 最后到过的位置，这个位置是倒数第二个位置（或最后的位置，这依赖于精确的实现方法）。这样的做法将会产生两个非常不平衡的子数组。如果所有的元素都是相同的，那么运行时间则是 $O(N^2)$ 。对于预排序的输入而言，其效果与使用第一个元素作为枢纽元相同。它花费的时间是二次的，可是却什么事也没做！

这样我们就发现，进行不必要的交换建立两个均衡的子数组比蛮干冒险得到两个不平衡的子数组要好。因此，如果 $i$ 和 $j$ 遇到等于枢纽元的元素，那么就让 $i$ 和 $j$ 都停止。对于这种输入，这实际上是不花费二次时间的四种可能性中唯一的一种可能。

初看起来，过多考虑具有相同元素的数组似乎有些愚蠢。难道有人偏要对50 000个相同的元素排序吗？为什么？我们记得，快速排序是递归的。设有1 000 000个元素，其中有50 000个是相同的（或更可能的情况是其排序键都相同的复杂元素）。最后，快速排序将对这50 000个元素进行递归调用。此时，真正重要的是确保这50 000个相同的元素能够被有效地排序。

### 7.7.3 小数组

对于很小的数组（ $N \leq 20$ ），快速排序不如插入排序好。而且，因为快速排序是递归的，所以这样的情形经常发生。通常的解决方法是，对于小的数组不递归地使用快速排序，而代之以诸如插入排序这样的对小数组有效的排序算法。使用这种策略实际上可以节省大约15%（相对于自始至终使用快速排序时）的运行时间。一种好的截止范围（cutoff range）是 $N = 10$ ，虽然在5~20之间任一截止范围都有可能产生类似的结果。这种做法也避免了一些有害的退化情形，如取三个元素的中值而实际上却只有一个或两个元素的情况。

### 7.7.4 实际的快速排序例程

快速排序的驱动程序见图7-14。

```

1  /**
2   * Quicksort algorithm (driver).
3   */
4  template <typename Comparable>
5  void quicksort( vector<Comparable> & a )
6  {
7      quicksort( a, 0, a.size( ) - 1 );
8  }
```

图7-14 快速排序的驱动程序

这种例程的一般形式是传递数组以及被排序数组的范围（left和right）。要处理的第一个

例程是枢纽元的选取。选取枢纽元最容易的方法是对`a[left]`、`a[right]`、`a[center]`适当地排序。这种方法还有额外的好处，即该三元素中的最小者被分在`a[left]`，而这正是分割阶段应该将它放到的位置。三元素中的最大者被分在`a[right]`，这也是正确的位置，因为它大于枢纽元。因此，可以把枢纽元放到`a[right-1]`并在分割阶段将`i`和`j`初始化到`left+1`和`right-2`。因为`a[left]`比枢纽元小，所以将它用作`j`的警戒标记，这是另一个好处。因此，我们不必担心`j`越界。由于`i`将停在那些等于枢纽元的元素处，故将枢纽元存储在`a[right-1]`则提供一个警戒标记。图7-15中的程序进行三数中值分割，它具有所描述的一切副作用。似乎使用实际上不对`a[left]`、`a[right]`、`a[center]`排序的方法计算枢纽元只不过效率稍微降低一些，但是很奇怪，这将产生不良结果（见练习7.46）。 284

```

1  /**
2   * Return median of left, center, and right.
3   * Order these and hide the pivot.
4   */
5  template <typename Comparable>
6  const Comparable & median3( vector<Comparable> & a, int left, int right )
7  {
8      int center = ( left + right ) / 2;
9      if( a[ center ] < a[ left ] )
10         swap( a[ left ], a[ center ] );
11      if( a[ right ] < a[ left ] )
12         swap( a[ left ], a[ right ] );
13      if( a[ right ] < a[ center ] )
14         swap( a[ center ], a[ right ] );
15
16         // Place pivot at position right - 1
17         swap( a[ center ], a[ right - 1 ] );
18         return a[ right - 1 ];
19 }

```

图7-15 执行三数中值分割方法的程序

图7-16的程序是快速排序真正的核心。它包括分割和递归调用。这里有几件事值得注意。第16行将`i`和`j`初始化为比它们的正确值超过1，使得不存在需要考虑的特殊情况。此处的初始化依赖于三数中值分割法有一些副作用的事实；如果按照简单的枢纽元策略使用该程序而不进行修正，那么这个程序是不能正确运行的，原因在于`i`和`j`开始于错误的位置而不再存在`j`的警戒标记。

第22行的交换动作为了速度上的考虑有时显式地写出。为使算法速度快，需要迫使编译器以直接插入的方式编译这些代码。如果`swap`是用`inline`声明的，则许多编译器都将自动这么做，但对于不这么做的编译器，差别可能会很明显。 285

最后，从第19行和第20行可以看出为什么快速排序这么快。算法的内部循环由一个增1/减1运算（运算很快）、一个测试以及一个转移组成。该算法没有像归并排序中那样的额外技巧，不过，这个程序仍然出奇地机敏。颇具诱惑力的做法是将第16行到第25行用图7-17中的语句代替，不过这是不能正确运行的，因为若`a[i]=a[j]=pivot`，则会产生一个无限循环。 286

### 7.7.5 快速排序的分析

正如归并排序那样，快速排序也是递归的，因此，它的分析要求解一个递推公式。我们将对快速排序进行这种分析，假设有一个随机的枢纽元（不用三数中值分割法）并对一些小的文件不设截止范围。和归并排序一样，取 $T(0) = T(1) = 1$ ，快速排序的运行时间等于两个递归调用的运

行时间加上花费在分割上的线性时间（枢纽元的选取仅花费常数时间）。我们得到基本的快速排序关系：

$$T(N) = T(i) + T(N-i-1) + cN \quad (7-1)$$

其中， $i = |S_1|$  是  $S_1$  中的元素个数。我们将考察三种情况。

```

1  /**
2   * Internal quicksort method that makes recursive calls.
3   * Uses median-of-three partitioning and a cutoff of 10.
4   * a is an array of Comparable items.
5   * left is the left-most index of the subarray.
6   * right is the right-most index of the subarray.
7   */
8  template <typename Comparable>
9  void quicksort( vector<Comparable> & a, int left, int right )
10 {
11     if( left + 10 <= right )
12     {
13         Comparable pivot = median3( a, left, right );
14
15         // Begin partitioning
16         int i = left, j = right - 1;
17         for( ; ; )
18         {
19             while( a[ ++i ] < pivot ) { }
20             while( pivot < a[ --j ] ) { }
21             if( i < j )
22                 swap( a[ i ], a[ j ] );
23             else
24                 break;
25         }
26
27         swap( a[ i ], a[ right - 1 ] ); // Restore pivot
28
29         quicksort( a, left, i - 1 ); // Sort small elements
30         quicksort( a, i + 1, right ); // Sort large elements
31     }
32     else // Do an insertion sort on the subarray
33         insertionSort( a, left, right );
34 }

```

图7-16 快速排序的主例程

```

16     int i = left + 1, j = right - 2;
17     for( ; ; )
18     {
19         while( a[ i ] < pivot ) i++;
20         while( pivot < a[ j ] ) j--;
21         if( i < j )
22             swap( a[ i ], a[ j ] );
23         else
24             break;
25     }

```

图7-17 对快速排序的小改动，它将中断该算法

### 1. 最坏情形的分析

枢纽元始终是最小元素。此时  $i = 0$ ，如果我们忽略无关紧要的  $T(0) = 1$ ，那么递推关系为

$$T(N) = T(N-1) + cN, \quad N > 1 \quad (7-2)$$

反复使用方程 (7-2), 得到

$$T(N-1) = T(N-2) + c(N-1) \quad (7-3)$$

$$T(N-2) = T(N-3) + c(N-2) \quad (7-4)$$

$$\vdots$$

$$T(2) = T(1) + c(2) \quad (7-5)$$

将所有这些方程相加, 得到

$$T(N) = T(1) + c \sum_{i=2}^N i = O(N^2) \quad (7-6)$$

这正是我们前面宣称的结果。

287

## 2. 最佳情形的分析

在最佳情形下, 枢纽元正好位于中间。为了简化数学推导, 我们假设两个子数组恰好各为原数组的一半大小, 虽然这会给出稍微过高的估计, 但是由于我们只关心大O答案, 因此结果还是可以接受的。

$$T(N) = 2T(N/2) + cN \quad (7-7)$$

用N去除方程 (7-7) 的两边, 得到

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c \quad (7-8)$$

反复套用这个方程, 得到

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + c \quad (7-9)$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + c \quad (7-10)$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c \quad (7-11)$$

将从式 (7-8) 到式 (7-11) 的方程加起来, 并注意, 它们共有  $\log N$  个, 于是

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c \log N \quad (7-12)$$

由此得到

$$T(N) = cN \log N + N = O(N \log N) \quad (7-13)$$

注意, 这和归并排序的分析完全相同, 因此, 得到相同的答案。

## 3. 平均情形的分析

这是最困难的部分。对于平均情形, 我们假设对于  $S_1$ , 每一个文件的大小都是等可能的, 因此每个大小均有概率  $1/N$ 。这个假设对于这里的枢纽元选取和分割方法实际上是合理的, 不过, 对于某些其他情况可能并不合理。那些不保持子数组随机性的分割方法不能使用这种分析方法。有趣的是, 这些方法看来导致程序在实际运行中花费更长的时间。

由该假设可知,  $T(i)$  (从而  $T(N-i-1)$ ) 的平均值为  $(1/N) \sum_{j=0}^{N-1} T(j)$ 。此时式 (7-1) 变成



288

$$T(N) = \frac{2}{N} \left[ \sum_{j=0}^{N-1} T(j) \right] + cN \quad (7-14)$$

如果用 $N$ 乘以方程(7-14), 则有

$$NT(N) = 2 \left[ \sum_{j=0}^{N-1} T(j) \right] + cN^2 \quad (7-15)$$

我们需要除去求和符号以简化计算。注意, 可以再套用一次式(7-15), 得到

$$(N-1)T(N-1) = 2 \left[ \sum_{j=0}^{N-2} T(j) \right] + c(N-1)^2 \quad (7-16)$$

若从式(7-15)减去式(7-16), 则得到

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c \quad (7-17)$$

移项、合并并除去右边无关紧要的项 $-c$ , 得到

$$NT(N) = (N+1)T(N-1) + 2cN \quad (7-18)$$

现在我们有了一个只用 $T(N-1)$ 表示 $T(N)$ 的公式。再用叠缩公式的思路, 不过式(7-18)的形式不适合。为此, 用 $N(N+1)$ 除式(7-18):

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1} \quad (7-19)$$

现在进行叠缩:

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N} \quad (7-20)$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1} \quad (7-21)$$

$$\vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \quad (7-22)$$

将式(7-19)到式(7-22)相加, 得到

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} \frac{1}{i} \quad (7-23)$$

该和大约为 $\log_e(N+1) + \gamma - 3/2$ , 其中  $\gamma \approx 0.577$ , 叫作欧拉常数 (Euler's constant), 于是

$$\frac{T(N)}{N+1} = O(\log N) \quad (7-24)$$

从而

$$T(N) = O(N \log N) \quad (7-25)$$

虽然这里的分析看似复杂, 但是实际上并不复杂——一旦你看出某些递推关系, 这些步骤是很自然的。该分析实际上还可以再进一步。上面描述的高度优化形式也已经分析过了, 结果的获得非常困难, 涉及一些复杂的递归和高深的数学。相等元素的影响也已仔细地进行了分析, 实际上所介绍的程序就是这么做的。

289

### 7.7.6 选择问题的线性期望时间算法

可以修改快速排序以解决选择问题 (selection problem)，这种问题我们在第1章和第6章已经见过。当时，通过使用优先队列，我们能够以时间  $O(N + k \log N)$  找到第  $k$  个最大（或最小）元。对于查找中值的特殊情况，它给出一个  $O(M \log N)$  算法。

由于我们能够以  $O(M \log N)$  时间给数组排序，因此可以期望为选择问题得到一个更好的时间界。我们介绍的查找集合  $S$  中第  $k$  个最小元的算法几乎与快速排序相同。事实上，其前三步是一样的。我们把这种算法叫作快速选择 (quickselect)。令  $|S_i|$  为  $S_i$  中元素的个数，快速选择的步骤如下：

(1) 如果  $|S|=1$ ，那么  $k=1$  并将  $S$  中的元素作为答案返回。如果正在使用小数组的截止方法且  $|S| \leq \text{CUTOFF}$ ，则将  $S$  排序并返回第  $k$  个最小元。

(2) 选取一个枢纽元  $v \in S$ 。

(3) 将集合  $S - \{v\}$  分割成  $S_1$  和  $S_2$ ，就像快速排序中所做的那样。

(4) 如果  $k \leq |S_1|$ ，那么第  $k$  个最小元必然在  $S_1$  中。在这种情况下，返回  $\text{quickselect}(S_1, k)$ 。如果  $k = 1 + |S_1|$ ，那么枢纽元就是第  $k$  个最小元，将它作为答案返回。否则，第  $k$  个最小元就在  $S_2$  中，它是  $S_2$  中的第  $(k - |S_1| - 1)$  个最小元。我们进行一次递归调用并返回  $\text{quickselect}(S_2, k - |S_1| - 1)$ 。

与快速排序对比，快速选择只进行了一次递归调用而不是两次。快速选择的最坏情形和快速排序的相同，也是  $O(N^2)$ 。直观看来，这是因为快速排序的最坏情形发生在  $S_1$  和  $S_2$  有一个是空的时候；于是，快速选择也就不是真的节省一次递归调用。不过，平均运行时间是  $O(N)$ 。具体分析类似于快速排序的分析，我们将它留作练习。

快速选择的实现甚至比抽象描述还要简单，其程序见图7-18。当算法终止时，第  $k$  个最小元就在位置  $k-1$  上（因为数组开始于下标0）。这破坏了原来的排序；如果不希望这样，那么需要做一个备份。

使用三数中值选取枢纽元的方法使得最坏情形发生的机会几乎是微不足道的。然而，通过仔细选择枢纽元，我们可以消除二次的最坏情形而保证算法是  $O(N)$  的。可是这么做的额外开销是相当大的，因此最终的算法主要在于理论上的意义。在第10章我们将考察选择问题的线性时间最坏情形算法，还将看到选取枢纽元的一个有趣的技巧，它使得选择算法在实践中多少要快一些。

290

## 7.8 间接排序

快速排序就是快速排序，谢尔排序就是谢尔排序。然而，直接应用基于这些算法的函数模板时，如果要排序的 Comparable 对象很大的话，有时效率会很低。问题就在于，在重新排列 Comparable 时，进行了太多的复制 Comparable 的工作（通过调用相应的 operator= 函数）。如果 Comparable 对象很大而且难于复制的话，其代价也会很高。

一般来说，这个问题的解决方案很简单：生成一个指向 Comparable 的指针数组，然后重新排列这些指针。一旦确定了元素应该在的位置，就可以直接将该元素放在相应的位置上，而不必进行过多的中间复制操作。这需要使用称为中间置换 (in-situ permutation) 的算法。用 C++ 实现的时候需要一些新的语法。

算法的第一步生成一个指针数组。令  $a$  为要排序的数组， $p$  为指针数组。初始化后， $p[i]$  指向  $a[i]$  中的对象。下一步，使用  $p[i]$  所指向对象的值来确定  $p[i]$  的顺序并对  $p[i]$  进行排序。在数组  $a$  中的对象完全不动，而数组  $p$  中的指针则重新排列。图7-19给出了排序后的指针数组。

```

1  /**
2   * Internal selection method that makes recursive calls.
3   * Uses median-of-three partitioning and a cutoff of 10.
4   * Places the kth smallest item in a[k-1].
5   * a is an array of Comparable items.
6   * left is the left-most index of the subarray.
7   * right is the right-most index of the subarray.
8   * k is the desired rank (1 is minimum) in the entire array.
9   */
10 template <typename Comparable>
11 void quickSelect( vector<Comparable> & a, int left, int right, int k )
12 {
13     if( left + 10 <= right )
14     {
15         Comparable pivot = median3( a, left, right );
16
17         // Begin partitioning
18         int i = left, j = right - 1;
19         for( ; ; )
20         {
21             while( a[ ++i ] < pivot ) { }
22             while( pivot < a[ --j ] ) { }
23             if( i < j )
24                 swap( a[ i ], a[ j ] );
25             else
26                 break;
27         }
28
29         swap( a[ i ], a[ right - 1 ] ); // Restore pivot
30
31         // Recurse; only this part changes
32         if( k <= i )
33             quickSelect( a, left, i - 1, k );
34         else if( k > i + 1 )
35             quickSelect( a, i + 1, right, k );
36     }
37     else // Do an insertion sort on the subarray
38         insertionSort( a, left, right );
39 }

```

图7-18 快速选择的主例程

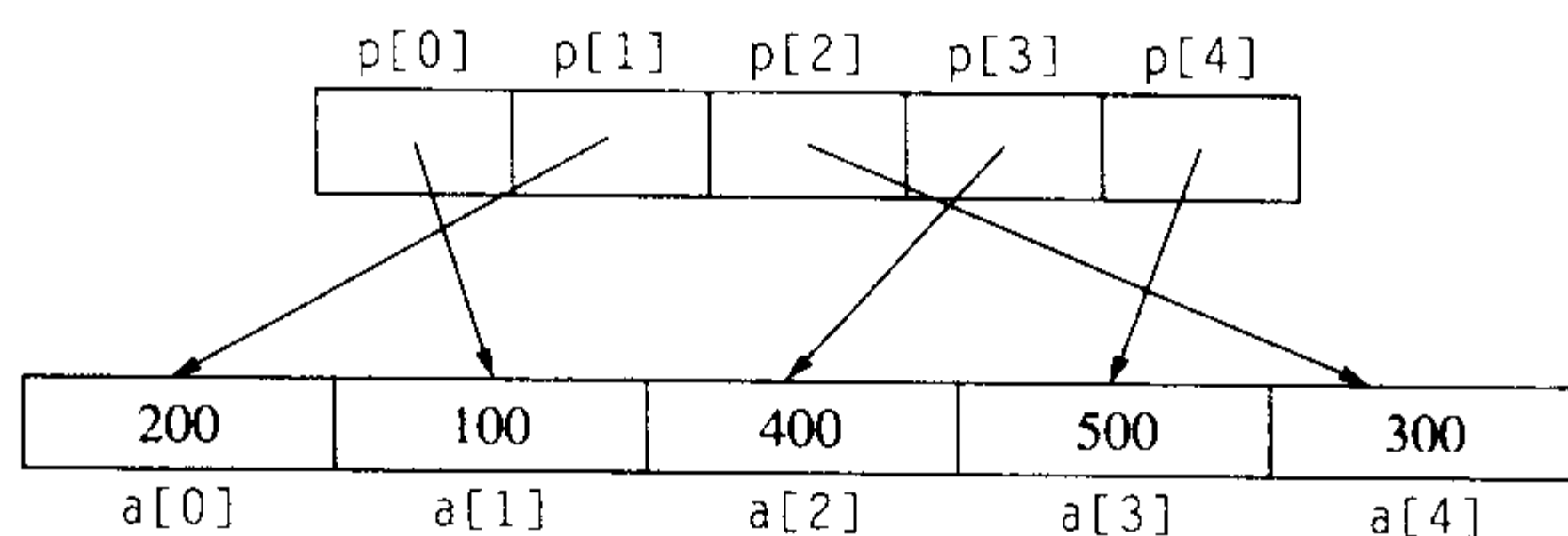


图7-19 使用指针数组排序

我们还是需要重新排列数组a。最简单的方法是定义第二个Comparable数组，称之为copy。然后就可以按正确的顺序将对象写入copy，然后再将copy写回a。这么做的代价是使用一个额外的数组，并且需要使用总计 $2N$ 次Comparable复制。

该算法有一个潜在的严重问题。使用copy导致了双倍的空间需求。假设 $N$ 很大（否则就可以使用插入排序）并且Comparable对象也很大（否则就不必使用指针来实现了）。这样，我们有理由相信这个操作几乎是运行在计算机内存的极限范围。虽然我们可以期望使用指针的一个额外的

vector向量，但是不会寄希望于Comparable对象的额外的vector向量可用。因此，我们需要在不使用额外数组的情况下将数组a进行重新排列。

随之而来的第二个问题是在使用copy时，总计使用了 $2N$ 次的Comparable复制。虽然这相对于原始的算法而言已经是很大的改进了，我们还是可以对该算法进行进一步的改进。特别地，我们将不会使用超过 $3N/2$ 次的Comparable复制，并且，对于所有的输入，只需要使用比 $N$ 稍多一点的复制。我们不仅会节省空间，也会节省时间。在进一步讨论代码前，我们先对所要做的进行概括说明。令人吃惊的是在这之前我们已经这么做了。

为说明我们的意图，首先从 $i=2$ 开始。由于 $p[2]$ 指向 $a[4]$ ，可知，需要将 $a[4]$ 移动到 $a[2]$ 。首先，需要将 $a[2]$ 存储起来，否则以后就不能正确地移动 $a[2]$ 。令 $tmp=a[2]$ ，然后 $a[2]=a[4]$ 。当 $a[4]$ 移到 $a[2]$ 后，就可以将一个本质为空的对象移动到 $a[4]$ 。观察 $p[4]$ ，可以看到，正确的表达式是 $a[4]=a[3]$ 。接下来，我们需要移动对象到 $a[3]$ 中。由于 $p[3]$ 指向 $a[2]$ ，可知，需要将 $a[2]$ 移动到此处。但是 $a[2]$ 已经在开始重新排列的时候被覆盖了。鉴于 $a[2]$ 的初始值为 $tmp$ ，有 $a[3]=tmp$ 。这个过程显示了从 $i=2$ 开始，遵循 $p$ 数组，形成了一个循环序列2,4,3,2:

```
tmp = a[2];
a[2] = a[4];
a[4] = a[3];
a[3] = tmp;
```

我们重新排列了三个元素，却仅仅使用了四次Comparable复制和一个额外的Comparable存储。事实上，之前我们已经使用了这个方法。最内层的插入循环存储当前的 $a[i]$ 在 $tmp$ 对象中。然后进行赋值 $a[j]=a[j-1]$ 来进行大量的元素的移动。最后，通过 $a[j]=tmp$ 放回原始的元素。这里我们的做法完全一样，唯一不同之处在于，这里不是一个一个地依次访问元素，而是使用 $p$ 来引导重新排列的执行。在二叉堆insert中也使用了相同的滑动算法。

一般来说，我们会有许多重新排列所需循环的集合。在图7-21中，有两个循环，一个包含两个元素，另一个包含三个元素。重新排列长为 $L$ 的循环需要使用 $L+1$ 次Comparable复制。长度为1的循环只有一个已经正确排序的元素，因此不需要复制。这改进了前面的算法，因为现在对于已经排序的数组不需要任何的Comparable复制。

对给定 $N$ 个元素的数组，令 $C_L$ 是长度为 $L$ 的循环的次数。Comparable复制次数 $M$ 如下：

$$M = N - C_1 + (C_2 + C_3 + \dots + C_N) \quad (7-26)$$

最好的情况就是没有Comparable复制，此时有 $N$ 个长为1的循环（也就是说，每一个元素都已经在正确的位置了）。最坏的情况为有 $N/2$ 个长度为2的循环。在这种情况下式（7-26）指出需要执行 $M=3N/2$ 次Comparable复制。当输入为2, 1, 4, 3, 6, 5等，这种情况就会发生。希望的 $M$ 值是什么？在练习中给出这个 $M$ 值是 $N-2+H_N$ 。

为实现这个算法，首先为 $p$ 中所存储对象的类型提供一个类模板Pointer，如图7-20所示。然后如图7-21所示，编写一个名字为largeObjectSort的函数模板。这段代码使用了C++中关于指针处理的几个高级概念。

### 7.8.1 vector<Comparable\*>不运行

基本的思想就是声明指针数组 $p$ 为 $vector<Comparable*>$ ，然后调用 $quicksort(p)$ 来重新排列指针。但是这不能运行。其问题就在于，对 $quicksort$ 的模板算法为 $Comparable*$ ，因此我们需要能够比较两个 $Comparable*$ 类型的“<”操作符。这样的操作符对于指针变量来说是存在的（回想1.5.1节）。但是这个操作符对指针所指向的Comparable所存储的值却什么都不做。此外，

291  
292

293



这种情况也改变不了。

```

1  template <typename Comparable>
2  class Pointer
3  {
4  public:
5      Pointer( Comparable *rhs = NULL ) : pointee( rhs ) { }
6
7      bool operator<( const Pointer & rhs ) const
8      { return *pointee < *rhs.pointee; }
9
10     operator Comparable * ( ) const
11     { return pointee; }
12 private:
13     Comparable *pointee;
14 };

```

图7-20 存储指向Comparable的指针的类

```

15 template <typename Comparable>
16 void largeObjectSort( vector<Comparable> & a )
17 {
18     vector<Pointer<Comparable> > p( a.size( ) );
19     int i, j, nextj;
20
21     for( i = 0; i < a.size( ); i++ )
22         p[ i ] = &a[ i ];
23
24     quicksort( p );
25
26     // Shuffle items in place
27     for( i = 0; i < a.size( ); i++ )
28         if( p[ i ] != &a[ i ] )
29         {
30             Comparable tmp = a[ i ];
31             for( j = i; p[ j ] != &a[ i ]; j = nextj )
32             {
33                 nextj = p[ j ] - &a[ 0 ];
34                 a[ j ] = *p[ j ];
35                 p[ j ] = &a[ j ];
36             }
37             a[ j ] = tmp;
38             p[ j ] = &a[ j ];
39         }
40 }

```

图7-21 对大对象进行排序的算法

## 7.8.2 智能指针类

这个问题的解决方案是声明一个新的类模板，Pointer<sup>1</sup>。Pointer将一个指向Comparable的指针作为数据成员来存储。然后就可以为Pointer类型添加一个比较操作符。这与图1-21中的Employee类相似。

数据成员pointee在第13行的私有部分声明。Pointer的构造函数需要给Pointee赋一个初始值（或NULL），见第5行。

包含指针动作的类有时候也称为**智能指针类**（smart pointer classes）。这个类比普通的指针要

1. 另外一个选择是，如果排序例程允许，则使用函数对象。函数对象方法对正常标准库排序可用。

高级一点，因为如果没有赋初始值，它可以自动地将自身初始化为NULL。

### 7.8.3 重载operator<

实现operator<在概念上来说是很简单的。我们仅仅需要将<操作符应用在所指向的Comparable对象就可以了。特别注意，这里不是递归逻辑。第7行中类Pointer的operator<（模板）比较两个Pointer类型。第8行的调用比较两个Comparable类型。

### 7.8.4 使用“\*”解引用指针

第8行中的“\*”是什么？在C++中“\*”是指针的解引用运算符。如果ptr是一个指向对象的指针，那么\*ptr就是所指向对象的同义词。换句话说，如果ptr指向对象obj，那么\*ptr与obj相同。\*ptr的值就是obj的值，而且任何对\*ptr的改变都影响obj。

该操作符和取地址操作符“&”带来了比C++中的其他操作符多得多的麻烦。但是，它们是必需的，而且在这里是不可避免的。

### 7.8.5 重载类型转换操作符

第10行很好地显示了C++语法的奇特性。这是类型转换操作符，特别地，这种方法定义了一个从Pointer<Comparable>到Comparable\*的类型转换。该实现很简单，只需要在第11行返回Pointee。这允许我们获得指针。虽然也可以使用一个命名成员函数，例如getPointee，这个类型转换还是简化了largeObjectSort算法。

294  
295

### 7.8.6 随处可见的隐式类型转换

C++是完全的录入语言。在我们的代码中有如下的类型：

```
a[i]    Comparable
&a[i]   Comparable*
p[i]    Pointer<Comparable>
```

因此，我们期望&a[i]和p[i]不是类型兼容的。在代码中还有数不清的这样的例子，例如下面的4个不同的例子（其他的三个与从22行到28行的部分都是完全相同的）：

```
22    p[i]=&a[i];
28    if(p[i]!=&a[i])
33        nextj=p[j]-&a[0];
34        a[j]=*p[j];
```

对于完全的录入来说这意味着什么？在第10行，我们通过提供类型转换操作符来禁止隐式类型转换，同时没有对Pointer构造函数使用explicit。下面看一下具体的例子。

第22行可以正常工作，因为Pointer构造函数不是explicit的。既然p[i]是Pointer<Comparable>的，右侧的也必须一样。尽管如此，它也不是explicit的，可以使用Pointer<Comparable>构造函数来构造一个临时变量，这意味着后台构造是允许的，因为explicit被省略了。

第28行使用operator!=来比较Pointer<Comparable>和Comparable\*。该操作不可实现。然而，隐式的类型转换（在第10行使用类型转换操作符）可以用来生成一个临时变量Comparable\*。相应地，operator!=成立。

第33行是C++的另一个小技巧。后面我们将要讨论这个问题。这里，第10行的类型转换操作

符从`p[j]`中生成了一个临时`Comparable*`。然而，这个操作符不是为该类型定义的（该操作符可以重载，但是我们一直没有这么做）。第10行的类型转换操作符通过从`p[j]`中生成一个临时的`Comparable*`来存储日期。

### 7.8.7 双向隐式类型转换会导致歧义

这些类型转换在使用的时候功能很强大，但是也会引起一些意想不到的问题。例如，假设对第7行和第8行定义的`operator<`，附加`operator!=`。（这不是很大的扩展；某些高级查找树依赖附加给`operator<`的`operator!=`。）

现在第28行不再编译了。这是因为此处生成了一个歧义。现在既可以转换`p[i]`为`Comparable*`，并使用为基本指针变量定义的`!=`操作符，又可以转换`&a[i]`为`Pointer<Comparable>`，使用构造函数然后使用为`Pointer<Comparable>`定义的`!=`操作符。

有许多方法可以步出这个困境。但是只要不在任何重要的类中定义双向隐式转换就足够了。  
296 如果总是使用`explicit`或者总也不使用类型转换操作符，那么就永远也不会遇到这个问题。

### 7.8.8 指针减法是合法的

最后一个奇特的地方在第33行。如果`p1`和`p2`指向同一个数组中的两个元素，那么`p1-p2`是它们分开的距离。该距离为`int`型的。因此，`p[j]-&a[0]`是`p[j]`所指向对象的索引。

## 7.9 排序算法的一般下界

虽然我们得到一些 $O(M\log N)$ 的排序算法，但是，尚不清楚我们是否还能做得更好。本节我们证明，任何只用到比较的排序算法在最坏情形下需要 $\Omega(M\log N)$ 次比较，因此归并排序和堆排序在一个常数因子范围内是最优的。该证明可以扩展到证明对只用到比较的任何排序算法都需要 $\Omega(M\log N)$ 次比较，甚至平均情形也是如此。这意味着，快速排序在相差一个常数因子的范围内平均是最优的。

特别地，我们将证明下列结果：只用到比较的任何排序算法在最坏情形下都需要 $\lceil \log(N!) \rceil$ 次比较并平均需要 $\log(N!)$ 次比较。我们将假设，所有 $N$ 个元素是互异的，因为任何排序算法都必须在这种情况下正常运行。

### 决策树

**决策树**（decision tree）是用于证明下界的抽象过程。在这里，决策树是一棵二叉树。每个结点表示元素之间一组可能的排序，它与已经进行的比较相一致。比较的结果是树的边。

图7-22中的决策树表示将三个元素 $a$ 、 $b$ 和 $c$ 排序的算法。算法的初始状态在根处（我们将可互换地使用术语状态和结点）。没有进行比较，因此所有的顺序都是合法的。这个特定的算法进行的第一次比较是比较 $a$ 和 $b$ 。两种比较的结果导致两种可能的状态。如果 $a < b$ ，那么只有三种可能性被保留。如果算法到达结点2，那么它将比较 $a$ 和 $c$ 。其他算法可能会做不同的工作；不同的算法可能有不同的决策树。若 $a > c$ ，则算法进入状态5。由于只存在一种相容的顺序，因此算法可以终止并报告它已经完成了排序。若 $a < c$ ，则算法尚不能终止，因为存在两种可能的顺序，它还不能肯定哪种是正确的。在这种情况下，算法还将再需要一次比较。

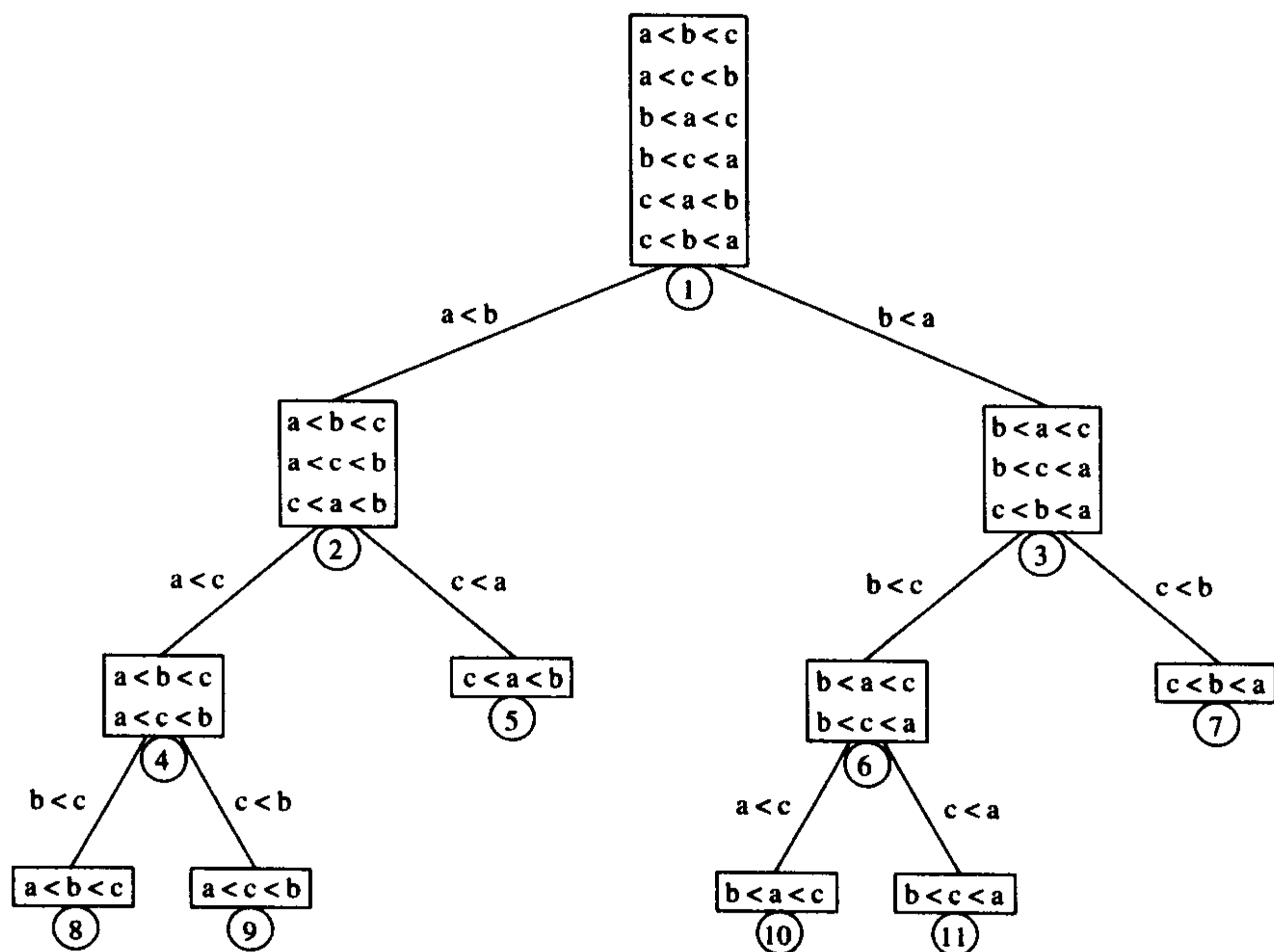


图7-22 三元素排序的决策树

通过只使用比较进行排序的每一种算法都可以用决策树表示。当然，只有输入数据非常少的情况画决策树才是可行的。由排序算法所使用的比较次数等于最深的树叶的深度。在我们的例子中，该算法在最坏情形下使用了三次比较。所使用的比较的平均次数等于树叶的平均深度。由于决策树很大，因此必然存在一些长的路径。为了证明下界，需要证明某些基本的树性质。

297

**引理7.1** 令 $T$ 是深度为 $d$ 的二叉树，则 $T$ 最多有 $2^d$ 片树叶。

**证明** 用数学归纳法证明。如果 $d=0$ ，则最多存在一片树叶，因此基准情形为真。若 $d>0$ ，则有一个根，它不可能是树叶，其左子树和右子树中每一个的深度最多是 $d-1$ 。由归纳假设，每一棵子树最多有 $2^{d-1}$ 片树叶，因此总数最多有 $2^d$ 片树叶。这就证明了该引理。 ■

**引理7.2** 具有 $L$ 片树叶的二叉树的深度至少是 $\lceil \log L \rceil$ 。

**证明** 由前面的引理立即推出。 ■

**定理7.6** 只使用元素间比较的任何排序算法在最坏情形下至少需要 $\lceil \log(N!) \rceil$ 次比较。

298

**证明** 对 $N$ 个元素排序的决策树必然有 $N!$ 片树叶。从上面的引理即可推出该定理。 ■

**定理7.7** 只使用元素间比较的任何排序算法需要 $\Omega(N \log N)$ 次比较。

**证明** 由前面的定理可知，需要 $\log(N!)$ 次比较。

$$\begin{aligned}
 \log(N!) &= \log(N(N-1)(N-2) \cdots (2)(1)) \\
 &= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
 &\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(N/2) \\
 &\geq \frac{N}{2} \log \frac{N}{2} \\
 &\geq \frac{N}{2} \log N - \frac{N}{2} \\
 &= \Omega(N \log N)
 \end{aligned}$$



这种类型的下界论断,当用于证明最坏情形结果时,有时叫作信息理论(information-theoretic)下界。这里的一般定理说的是,如果存在 $P$ 种不同的可能情况要区分,而问题是YES/NO的形式,那么通过任何算法求解该问题在某种情形下总需要 $\lceil \log P \rceil$ 个问题。对于任何基于比较的排序算法的平均运行时间,证明类似的结果是可能的。这个结果由下列引理导出,我们将它留作练习:具有 $L$ 片树叶的任意二叉树的平均深度至少为 $\log L$ 。

## 7.10 桶排序

虽然我们在上一节证明了,任何只使用比较的一般排序算法在最坏情形下需要 $\Omega(M \log N)$ 运行时间,但是记住,在某些特殊情况下以线性时间进行排序仍然是可能的。

一个简单的例子是桶排序(bucket sort)。为使桶排序能够正常工作,必须要有一些附加的信息。输入数据 $A_1, A_2, \dots, A_N$ , 必须只由小于 $M$ 的正整数组成(显然还可以对其进行扩充)。如果是这种情况,那么算法很简单:使用一个大小为 $M$ 的称为count的数组,它被初始化为全0。于是,count有 $M$ 个单元(或称桶),这些桶初始化为空。当读 $A_i$ 时,  $\text{count}[A_i]$ 增1。在所有的输入数据读入后,扫描数组count,打印出排序后的表。该算法用时 $O(M + N)$ ;其证明留作练习。如果 $M$ 为 $O(N)$ ,那么总量就是 $O(N)$ 。

299 虽然这个算法似乎干扰了下界,但事实上并没有,因为它使用了比简单比较更为强大的操作。通过使适当的桶增值,算法在单位时间内实质上执行了一个 $M$ 路比较。这类似于用在可扩散列上的策略(见5.7节)。显然这不属于那种下界已证明的模型。

不过,该算法确实提出了用于证明下界的模型的合理性问题。这个模型实际上是一个强模型,因为通用的排序算法不能对于它可以预期见到的输入类型做假设,但必须仅仅基于排序信息做一些决策。很自然地,如果存在额外的可用信息,我们应该有望找到更为有效的算法,否则这额外的信息就被浪费了。

尽管桶排序看似太一般而用处不大,但是实际上却存在许多其输入只是一些小的整数的情况,使用像快速排序这样的排序方法就是小题大做了。

## 7.11 外部排序

迄今为止,已经考察过的所有算法都需要将输入数据装入主存。然而,有一些应用程序,它们的输入数据量太大装不进内存。本节将讨论一些外部排序(external sorting)算法,它们是用来处理大量的输入的。

### 7.11.1 为什么需要新算法

大部分内部排序算法都用到内存可直接寻址的事实。谢尔排序用一个时间单位比较元素 $a[i]$ 和 $a[i-h_k]$ 。堆排序用一个时间单位比较元素 $a[i]$ 和 $a[i*2+1]$ 。使用三数中值分割法的快速排序以常数个时间单位比较 $a[\text{left}]$ 、 $a[\text{center}]$ 和 $a[\text{right}]$ 。如果输入数据在磁带上,那么所有这些操作就失去了它们的效率,因为磁带上的元素只能被顺序访问。即使数据在磁盘上,由于转动磁盘和移动磁头所需的延迟,仍然存在实际上的效率损失。

为了看到外部访问究竟有多慢,可建立一个大的随机文件,但不能太大以至装不进主存。将该文件读入并用一种有效的算法对其排序。将该输入数据进行排序所花费的时间与将其读入所花费的时间相比必然是无足轻重的,尽管排序是 $O(M \log N)$ 操作而读入数据只不过花费 $O(N)$ 时间。

### 7.11.2 外部排序模型

各种各样的海量存储装置使得外部排序比内部排序对设备的依赖性要严重得多。我们将考虑的一些算法在磁带上工作，而磁带可能是最受限制的存储介质。由于访问磁带上一个元素需要把磁带转动到正确的位置，因此磁带必须要有（两个方向上）连续的顺序才能够被有效地访问。

我们将假设至少有三个磁带驱动器进行排序工作，其中两个驱动器执行有效的排序，而第三个驱动器进行简化的工作。如果只有一个磁带驱动器可用，那么我们不得不说：任何算法都将需要 $\Omega(N^2)$ 次磁带访问。

300

### 7.11.3 简单算法

基本的外部排序算法使用归并排序中的合并算法。设有四盘磁带 $T_{a1}$ 、 $T_{a2}$ 、 $T_{b1}$ 和 $T_{b2}$ ，它们是两盘输入磁带和两盘输出磁带。根据算法的特点，磁带 $a$ 和磁带 $b$ 或者用作输入磁带，或者用作输出磁带。设数据最初在 $T_{a1}$ 上，并设内存可以一次容纳（和排序） $M$ 个记录。一种自然的做法是首先从输入磁带一次读入 $M$ 个记录，在内部将这些记录排序，然后再把排过序的记录交替地写到 $T_{b1}$ 或 $T_{b2}$ 上。我们将把每组排过序的记录叫作一个顺串（run）。做完这些之后，倒回所有的磁带。设这里的输入与谢尔排序的例子中的输入数据相同。

|          |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|
| $T_{a1}$ | 81 | 94 | 11 | 96 | 12 | 35 | 17 | 99 | 28 | 58 | 41 | 75 | 15 |
| $T_{a2}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{b1}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{b2}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |

如果 $M=3$ ，那么在上述顺串构造以后，磁带将包含下图所指出的数据。

|          |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|
| $T_{a1}$ |    |    |    |    |    |    |    |
| $T_{a2}$ |    |    |    |    |    |    |    |
| $T_{b1}$ | 11 | 81 | 94 | 17 | 28 | 99 | 15 |
| $T_{b2}$ | 12 | 35 | 96 | 41 | 58 | 75 |    |

现在 $T_{b1}$ 和 $T_{b2}$ 包含一组顺串。我们将每个磁带的第一个顺串取出并将二者合并，把结果写到 $T_{a1}$ 上，该结果是一个二倍长的顺串。注意，合并两个排过序的表是简单的操作，几乎不需要内存，因为合并是在 $T_{b1}$ 和 $T_{b2}$ 前进时进行的。然后，再从每盘磁带取出下一个顺串，合并，并将结果写到 $T_{a2}$ 上。继续这个过程，交替使用 $T_{a1}$ 和 $T_{a2}$ ，直到 $T_{b1}$ 或 $T_{b2}$ 为空。此时，或者 $T_{b1}$ 和 $T_{b2}$ 均为空，或者剩下一个顺串。对于后者，把剩下的顺串拷贝到适当的磁带上。将全部四盘磁带倒回，并重复相同的步骤，这一次用两盘 $a$ 磁带作为输入，两盘 $b$ 磁带作为输出，结果得到一些 $4M$ 的顺串。继续这个过程直到得到长为 $N$ 的一个顺串。

该算法将需要 $\lceil \log(N/M) \rceil$ 趟工作，外加一趟初始的顺串构造。例如，若有1000万个记录，每个记录128个字节，并有4 MB的内存，则第一趟将建立320个顺串。此时再需要九趟才能完成排序。我们的例子再需要 $\lceil \log 13/3 \rceil = 3$ 趟，见下图所示。

|          |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|
| $T_{a1}$ | 11 | 12 | 35 | 81 | 94 | 96 | 15 |
| $T_{a2}$ | 17 | 28 | 41 | 58 | 75 | 99 |    |
| $T_{b1}$ |    |    |    |    |    |    |    |
| $T_{b2}$ |    |    |    |    |    |    |    |

301

|          |    |    |    |    |    |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|
| $T_{a1}$ |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{a2}$ |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{b1}$ | 11 | 12 | 17 | 28 | 35 | 41 | 58 | 75 | 81 | 94 | 96 | 99 |
| $T_{b2}$ | 15 |    |    |    |    |    |    |    |    |    |    |    |

|          |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|
| $T_{a1}$ | 11 | 12 | 15 | 17 | 28 | 35 | 41 | 58 | 75 | 81 | 94 | 96 | 99 |
| $T_{a2}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{b1}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{b2}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |

#### 7.11.4 多路合并

如果有额外的磁带，那么可以减少将输入数据排序所需要的趟数，通过将基本的（2路）合并扩充为 $k$ 路合并就能做到这一点。

两个顺串的合并操作通过将每一个输入磁带转到每个顺串的开头来进行。然后，找到较小的元素，把它放到输出磁带上，并将相应的输入磁带向前推进。如果有 $k$ 盘输入磁带，那么这种方法以相同的方式工作，唯一的区别在于，它找到 $k$ 个元素中最小的元素的过程稍微复杂一些。可以使用优先队列找出这些元素中的最小元。为了得出下一个写到磁盘上的元素，进行一次deleteMin操作。将相应的磁带向前推进，如果输入磁带上的顺串尚未完成，那么将新元素insert到优先队列中。仍然利用前面的例子，将输入数据分配到三盘磁带上。

302

|          |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|
| $T_{a1}$ |    |    |    |    |    |    |
| $T_{a2}$ |    |    |    |    |    |    |
| $T_{a3}$ |    |    |    |    |    |    |
| $T_{b1}$ | 11 | 81 | 94 | 41 | 58 | 75 |
| $T_{b2}$ | 12 | 35 | 96 | 15 |    |    |
| $T_{b3}$ | 17 | 28 | 99 |    |    |    |

然后，还需要两趟3路合并以完成该排序。

|          |    |    |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|----|----|
| $T_{a1}$ | 11 | 12 | 17 | 28 | 35 | 81 | 94 | 96 | 99 |
| $T_{a2}$ | 15 | 41 | 58 | 75 |    |    |    |    |    |
| $T_{a3}$ |    |    |    |    |    |    |    |    |    |
| $T_{b1}$ |    |    |    |    |    |    |    |    |    |
| $T_{b2}$ |    |    |    |    |    |    |    |    |    |
| $T_{b3}$ |    |    |    |    |    |    |    |    |    |

|          |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|
| $T_{a1}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{a2}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{a3}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{b1}$ | 11 | 12 | 15 | 17 | 28 | 35 | 41 | 58 | 75 | 81 | 94 | 96 | 99 |
| $T_{b2}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |
| $T_{b3}$ |    |    |    |    |    |    |    |    |    |    |    |    |    |

在初始顺串构造阶段之后，使用 $k$ 路合并所需要的趟数为 $\lceil \log_k(N/M) \rceil$ ，因为在每趟合并中顺串达到 $k$ 倍大小。对于上面的例子，公式成立，因为 $\lceil \log_3 13/3 \rceil = 2$ 。如果有10盘磁带，那么 $k = 5$ ，而前一节的大例子需要的趟数将是 $\lceil \log_5 320 \rceil = 4$ 。



### 7.11.5 多相合并

上一节讨论的 $k$ 路合并方法需要使用 $2k$ 盘磁带，这对某些应用极为不便。只使用 $k+1$ 盘磁带也有可能完成排序的工作。作为例子，我们阐述只用三盘磁带如何完成2路合并。

设有三盘磁带 $T_1$ 、 $T_2$ 和 $T_3$ ，在 $T_1$ 上有一个输入文件，它将产生34个顺串。一种选择是在 $T_2$ 和 $T_3$ 的每一盘磁带中放入17个顺串。然后将结果合并到 $T_1$ 上，得到一盘有17个顺串的磁带。由于所有的顺串都在一盘磁带上，因此必须把其中的一些顺串放到 $T_2$ 上以进行另外的合并。执行该合并的逻辑方式是将前8个顺串从 $T_1$ 复制到 $T_2$ 并进行合并。这样的效果是对于我们所做的每一趟合并又附加了另外的半趟工作。

另一种选择是把原始的34个顺串不均衡地分成两份。设把21个顺串放到 $T_2$ 上而把13个顺串放到 $T_3$ 上。然后，将13个顺串合并到 $T_1$ 上直到 $T_3$ 用完。此时，可以倒回磁带 $T_1$ 和 $T_3$ ，然后将具有13个顺串的 $T_1$ 和8个顺串的 $T_2$ 合并到 $T_3$ 上。此时，合并8个顺串直到 $T_2$ 用完为止，这样，在 $T_1$ 上将留下5个顺串而在 $T_3$ 上则有8个顺串。然后，再合并 $T_1$ 和 $T_3$ ，等等。下面的图表显示了每趟合并之后每盘磁带上的顺串的个数。

|       | 顺串<br>个数 | $T_3 + T_2$<br>之后 | $T_1 + T_2$<br>之后 | $T_1 + T_3$<br>之后 | $T_2 + T_3$<br>之后 | $T_1 + T_2$<br>之后 | $T_1 + T_3$<br>之后 | $T_2 + T_3$<br>之后 |
|-------|----------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| $T_1$ | 0        | 13                | 5                 | 0                 | 3                 | 1                 | 0                 | 1                 |
| $T_2$ | 21       | 8                 | 0                 | 5                 | 2                 | 0                 | 1                 | 0                 |
| $T_3$ | 13       | 0                 | 8                 | 3                 | 0                 | 2                 | 1                 | 0                 |

顺串最初的分配造成了很大的不同。例如，若22个顺串放在 $T_2$ 上，12个在 $T_3$ 上，则第一趟合并后将得到 $T_1$ 上的12个顺串以及 $T_2$ 上的10个顺串。在下一次合并后， $T_1$ 上有10个顺串而 $T_3$ 上有2个顺串。此时，进展的速度慢了下来，因为在 $T_3$ 用完之前只能合并两组顺串。这时 $T_1$ 上有8个顺串而 $T_2$ 上有2个顺串。同样，只能合并两组顺串，结果 $T_1$ 上有6个顺串且 $T_3$ 上有2个顺串。再经过3趟合并之后， $T_2$ 上还有2个顺串而其余磁带均已没有任何内容。我们必须将一个顺串拷贝到另外一盘磁带上，然后结束合并。

事实上，我们给出的最初分配是最优的。如果顺串的个数是一个斐波那契数 $F_N$ ，那么分配这些顺串最好的方式是把它们分裂成两个斐波那契数 $F_{N-1}$ 和 $F_{N-2}$ 。否则，为了将顺串的个数补足成一个斐波那契数就必须用一些哑顺串（dummy run）来填补磁带。我们把如何将一组初始顺串分配到磁带上的具体做法留作练习。

可以把上面的做法扩充到 $k$ 路合并，此时我们需要 $k$ 阶斐波那契数用于分配顺串，其中 $k$ 阶斐波那契数定义为 $F^{(k)}(N) = F^{(k)}(N-1) + F^{(k)}(N-2) + \cdots + F^{(k)}(N-k)$ ，辅以适当的初始条件 $F^{(k)}(N) = 0, 0 \leq N \leq k-2, F^{(k)}(k-1) = 1$ 。

### 7.11.6 替换选择

最后我们将要考虑的是顺串的构造。迄今我们已经用到的策略是所谓的最简可能：读入尽可能多的记录并将它们排序，再把结果写到某个磁带上。这看起来像是可能的最佳处理，直到认识到：只要第一个记录被写到输出磁带上，它所使用的内存就可以被另外的记录使用。如果输入磁带上的下一个记录比刚刚输出的记录值大，那么它就可以被放入顺串中。

利用这种想法，可以给出产生顺串的一个算法，该方法通常称为替换选择（replacement selection）。开始， $M$ 个记录被读入内存并被放到一个优先队列中。我们执行一次deleteMin，把最小（值）的记录写到输出磁带上，再从输入磁带读入下一个记录。如果它比刚刚写出的记录大，



那么可以把它加到优先队列中，否则，不能把它放入当前的顺串。由于优先队列少一个元素，因此，可以把这个新元素存入优先队列的死区（dead space），直到顺串完成构建，而该新元素用于下一个顺串。将一个元素存入死区的做法类似于堆排序中的做法。继续这样的步骤直到优先队列的大小为零，此时该顺串构建完成。我们使用死区中的所有元素通过建立一个新的优先队列开始构建一个新的顺串。图7-23解释了这个小例子的顺串构建过程，其中 $M=3$ 。死元素以星号标示。

|      | 堆数组中的 3 个元素 |      |      | 输出   | 读的下一个元素 |
|------|-------------|------|------|------|---------|
|      | h[1]        | h[2] | h[3] |      |         |
| 顺串 1 | 11          | 94   | 81   | 11   | 96      |
|      | 81          | 94   | 96   | 81   | 12*     |
|      | 94          | 96   | 12*  | 94   | 35*     |
|      | 96          | 35*  | 12*  | 96   | 17*     |
|      | 17*         | 35*  | 12*  | 顺串结尾 | 重建堆     |
| 顺串 2 | 12          | 35   | 17   | 12   | 99      |
|      | 17          | 35   | 99   | 17   | 28      |
|      | 28          | 99   | 35   | 28   | 58      |
|      | 35          | 99   | 58   | 35   | 41      |
|      | 41          | 99   | 58   | 41   | 15*     |
|      | 58          | 99   | 15*  | 58   | 磁带结尾    |
|      | 99          |      | 15*  | 99   |         |
|      |             |      | 15*  | 顺串结尾 | 重建堆     |
| 顺串 3 | 15          |      |      | 15   |         |

图7-23 顺串构建的例子

在这个例子中，替换选择只产生3个顺串，这与通过排序得到5个顺串不同。正因为如此，3路合并经过一趟而非两趟而结束。如果输入数据是随机分配的，那么可以证明替换选择产生平均长度为 $2M$ 的顺串。对于我们所举的大例子，预计为160个顺串而不是320个顺串，因此，5路合并需要进行4趟。在这个例子中，我们没有节省一趟，虽然在幸运的情况下是可以节省的，并且可能有125个或更少的顺串。由于外部排序花费的时间太多，因此节省的每一趟都可能对运行时间产生显著的影响。

我们已经看到，替换选择做得可能并不比标准算法更好。然而，输入数据常常从排序或几乎都是从排序开始，此时替换选择仅仅产生少数非常长的顺串。这种类型的输入通常要进行外部排序，这就使得替换选择具有非常大的价值。

304

小结

使用C++，对于最一般的内部排序应用，选用的方法不是插入排序、谢尔排序，就是快速排序，它们的选用主要是根据输入的大小来决定。图7-24显示了每个算法（在一台相对较慢的计算机上）处理各种不同大小的输入时的运行时间。

这里，选择 $N$ 个整数组成一些随机排列，而给出的时间仅仅是排序的实际时间。图7-2给出的程序用于插入排序。谢尔排序使用7.4节中的程序，该程序改为使用Sedgewick增量运行。基于数以百万计次排序，大小从100到25 000 000不等，使用这种增量的谢尔排序的运行时间估计为

$O(N^{7/6})$ 。堆排序例程与7.5节中的相同。表中给出两种快速排序算法，第一种使用简单的枢纽元方法，不进行截止操作。幸运的是，输入是随机的。第二种使用三数中值分割法，截止范围为10。还可以进一步优化，比如可以写一个内嵌的三数中值例程而不是使用方法调用，也可以编写一个非递归的快速排序。还有其他一些方法可以对代码进行优化，但实现起来相当复杂，当然，也可使用汇编语言编程。我们已有打算高效地编写所有的例程，不过，性能因机器不同当然多少会有些变化。

| N       | 插入排序<br>$O(N^2)$ | 谢尔排序<br>$O(N^{7/6})(?)$ | 堆排序<br>$O(N \log N)$ | 快速排序<br>$O(N \log N)$ | 快速排序 (可选)<br>$O(N \log N)$ |
|---------|------------------|-------------------------|----------------------|-----------------------|----------------------------|
| 10      | 0.000001         | 0.000002                | 0.000003             | 0.000002              | 0.000002                   |
| 100     | 0.000106         | 0.000039                | 0.000052             | 0.000025              | 0.000023                   |
| 1000    | 0.011240         | 0.000678                | 0.000750             | 0.000365              | 0.000316                   |
| 10000   | 1.047            | 0.009782                | 0.010215             | 0.004612              | 0.004129                   |
| 100000  | 110.492          | 0.13438                 | 0.139542             | 0.058481              | 0.052790                   |
| 1000000 | NA               | 1.6777                  | 1.7967               | 0.6842                | 0.6154                     |

图7-24 不同的排序算法的比较（所有的时间均以秒计）

高度优化的快速排序算法即使对于很少的输入数据也能和谢尔排序一样快。快速排序的改进算法仍然有 $O(N^2)$ 的最坏情形（有一个练习让你构造一个小例子），但是，这种最坏情形出现的机会是微不足道的，以至于不能成为影响算法的因素。如果需要对大量数据排序，那么快速排序则是应该选择的方法。但是，永远都不要图省事而轻易地把第一个元素用作枢纽元。对输入数据是随机的假设并不安全。如果你不想过多地考虑这个问题，那么就使用谢尔排序。谢尔排序有些小缺陷，不过还是可以接受的，特别是需要简单明了的时候。谢尔排序的最坏情形也只不过是 $O(N^{4/3})$ ；这种最坏情形发生的几率也是微不足道的。

305

堆排序要比谢尔排序慢，尽管它是一个带有明显紧凑内循环的 $O(M \log N)$ 算法。对该算法的深入考察揭示出，为了移动数据，堆排序要进行两次比较。由Floyd提出的改进算法移动数据，基本上只需要一次比较，不过实现这种改进算法使得代码多少要长一些。我们把它留给读者来决定：这种附加的编程代价用以提高速度是否值得（练习7.51）。

插入排序只用在小的或是基本上排好序的输入数据上。我们没有提到归并排序，因为它在C++中的性能对于主存排序不如快速排序那么好，而且它的编程一点也不省事。然而我们已经看到，合并是外部排序的中心思想。

## 练习

- 7.1 使用插入排序将序列3, 1, 4, 1, 5, 9, 2, 6, 5排序。
- 7.2 如果所有的元素都相等，那么插入排序的运行时间是多少？
- 7.3 设我们交换元素 $a[i]$ 和 $a[i+k]$ ，它们最初是无序的。证明被去掉的逆序最少为1个、最多为 $2k-1$ 个。
- 7.4 写出使用增量{1, 3, 7}对输入数据9, 8, 7, 6, 5, 4, 3, 2, 1运行谢尔排序得到的结果。
- 7.5
  - a. 使用2增量序列{1, 2}的谢尔排序的运行时间是多少？
  - b. 证明，对任意的 $N$ ，存在一个3增量序列，使得谢尔排序以 $O(N^{5/3})$ 时间运行。
  - c. 证明，对任意的 $N$ ，存在一个6增量序列，使得谢尔排序以 $O(N^{3/2})$ 时间运行。
- 7.6 \*a. 证明，使用形如1,  $c$ ,  $c^2$ ,  $\dots$ ,  $c^i$ 的增量，谢尔排序的运行时间为 $\Omega(N^2)$ ，其中， $c$ 为任一整数。

306

**\*\*b.** 证明：对于这些增量，平均运行时间为 $\Theta(N^{3/2})$ 。

**\*7.7** 证明：若一个 $k$ -排序的文件被 $h$ -排序，则它仍是 $k$ -排序的。

**\*\*7.8** 证明：使用由Hibbard建议的增量序列的谢尔排序在最坏情形下的运行时间是 $\Omega(N^{3/2})$ 。提示：可以证明当所有的元素不是0就是1时谢尔排序在这种特殊情形下的时间界。如果 $i$ 可以表示为 $h_i, h_{i-1}, \dots, h_{\lfloor i/2 \rfloor + 1}$ 的线性组合，则可置 $a[i] = 1$ ，否则置为0。

**7.9** 确定谢尔排序对于下述情况的运行时间：

a. 排过序的输入数据。

**\*b.** 反序排列的输入数据。

**7.10** 下述两种对图7-6所编写的谢尔排序例程的修改影响最坏情形的运行时间吗？

a. 如果 $gap$ 是偶数，则在第8行前从 $gap$ 减1。

b. 如果 $gap$ 是偶数，则在第8行前往 $gap$ 加1。

**7.11** 指出堆排序如何处理输入数据142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102。

**7.12** 对于预排序的输入，堆排序的运行时间是多少？

**\*7.13** 证明存在这样的输入，它使得堆排序中的每一个`percolateDown`一直行进到树叶（提示：向后进行）。

**7.14** 重写堆排序，使得只对从`low`到`high`范围的项进行排序，其中`low`和`high`作为附加参数被传递。

**7.15** 用归并排序将3, 1, 4, 1, 5, 9, 2, 6排序。

**7.16** 不使用递归如何实现归并排序？

**7.17** 确定下列情况下归并排序的运行时间：

a. 已排序的输入数据。

b. 反序排列的输入数据。

c. 随机的输入数据。

**7.18** 在归并排序的分析中是不考虑常数的。证明，归并排序在最坏情形下用于比较的次数为 $N \lceil \log N \rceil - 2^{\lceil \log N \rceil} + 1$ 。

**7.19** 用三数中值分割法以及截止为3的快速排序将3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5排序。

**7.20** 使用本章中的快速排序实现方法确定下列输入数据的快速排序运行时间：

a. 已排序的输入数据。

b. 反序排列的输入数据。

c. 随机的输入数据。

**7.21** 当以下列元素作为枢纽元时，重做练习7.20：

a. 第一个元素。

b. 前两个互异元素中的较大者。

c. 一个随机元素。

**\*d.** 该输入集合中所有元素的平均值。

**7.22** a. 对于本章中快速排序的实现方法，当所有的键都相等时它的运行时间是多少？

b. 假设我们改变分割策略，使得当找到一个与枢纽元有相同键的元素时 $i$ 和 $j$ 都不停止。为了保证快速排序正常工作，需要对程序做哪些修改？当所有的键都相等时，运行时间是多少？

c. 假设我们改变分割策略，使得 $i$ 在一个与枢纽元有相同键的元素处停止，但是 $j$ 在类似的情形下却不停止。当所有的键都相等时，为了保证快速排序正常工作，需要对程序做哪些修改？快速排序的运行时间是多少？

**7.23** 设我们选择数组中间位置上的键作为枢纽元。这是否使得快速排序将不可能需要二次时间？

**7.24** 构造20个元素的一个排列，使得对于三数中值分割且截止为3的快速排序，该排列尽可能地差。

**7.25** 书中的快速排序使用两个递归调用。删除一个调用如下：



- a. 重写程序使得第二个递归调用无条件地成为快速排序的最后一行。通过将if/else的判断条件互换并在对insertionSort调用之后返回来做这一点。
- b. 通过写一个while循环并改变left来删除尾递归。
- 7.26 继续练习7.25a, 做以下工作:
- a. 执行一次测试, 使得较小的子数组由第一个递归调用处理, 而较大的子数组由第二个递归调用处理。
- b. 通过写一个while循环并在必要时改变left或right以删除尾递归。
- c. 证明递归调用的次数在最坏情形下是对数级。
- 7.27 设递归快速排序从驱动程序接收int型参数depth, 它的初始值近似为 $2\log N$ 。
- a. 修改递归快速排序使其在递归的层达到depth时对当前的子数组调用heapsort (提示: 当进行递归调用时depth减1; 当它为0时切换到堆排序)。
- b. 证明该算法的最坏情形运行时间为 $O(M\log N)$ 。
- c. 通过实验确定对heapsort调用的频率。
- d. 连同使用练习7.25中的删除尾递归一起, 实现本题的方法。
- e. 解释为什么练习7.26中的方法不再是必需的。
- 7.28 当实现快速排序时, 如果数组包含许多重复元, 那么可能更好的方法是执行3路划分 (划分成小于、等于以及大于枢纽元的三部分元素) 以进行更小的递归调用。假设使用3路比较。
- a. 给出一个算法, 该算法只使用 $N-1$ 次3路比较而将一个 $N$ 元素子数组实施3路原位划分。如果有 $d$ 项等于枢纽元, 那么可以使用 $d$ 次附加的Comparable交换, 高于和超越2路分割算法 (提示: 随着 $i$ 和 $j$ 彼此相向移动, 保持5组元素如下):

|       |       |         |       |       |
|-------|-------|---------|-------|-------|
| EQUAL | SMALL | UNKNOWN | LARGE | EQUAL |
|       | $i$   |         | $j$   |       |

308

- b. 证明: 使用上面的算法将只含有 $d$ 个不同值的 $N$ 元素数组排序花费 $O(dN)$ 时间。
- 7.29 编写一个程序实现选择算法。
- 7.30 求解下列递推关系:  $T(N) = (1/N) \left[ \sum_{i=0}^{N-1} T(i) \right] + cN$ ,  $T(0) = 0$ 。
- 7.31 如果一切具有相等键的元素都保持它们在输入数据时呈现的顺序, 那么这种排序算法就是稳定 (stable) 的。本章中的排序算法哪些是稳定的? 哪些不是? 为什么?
- 7.32 设给定 $N$ 个已排序的元素, 后面跟有 $f(N)$ 个随机顺序的元素。如果 $f(N)$ 是下列情况, 那么如何将全部数据排序?
- a.  $f(N) = O(1)$ 。
- b.  $f(N) = O(\log N)$ 。
- c.  $f(N) = O(\sqrt{N})$ 。
- \*d.  $f(N)$ 多大, 使得全部数据仍然能够以 $O(N)$ 时间排序?
- 7.33 证明: 在 $N$ 个元素已排序的表中找出一个元素 $X$ 的任何算法都需要 $\Omega(\log N)$ 次比较。
- 7.34 利用Stirling公式 $N! \approx (N/e)^N \sqrt{2\pi N}$ 给出 $\log(N!)$ 的精确估计。
- 7.35 \*a. 两个排过序的 $N$ 个元素的数组有多少种合并的方法?
- \*b. 给出合并两个 $N$ 个元素的排过序的表所需要的比较次数的非平凡下界。
- 7.36 考虑下列将6个数排序的算法:
- 使用算法A将前3个数排序。
  - 使用算法B将后3个数排序。
  - 使用算法C将两个已排序的数组合并。
- 证明这个算法是次优的 (suboptimal), 与算法A、B、C的选择无关。



- \*7.37 给出一个线性时间算法，将 $N$ 个分数排序，它们的分子和分母都是1和 $N$ 之间的整数。
- 7.38 设数组 $A$ 和 $B$ 都是已排序的并且均含有 $N$ 个元素。给出一个 $O(\log N)$ 算法找出 $A \cup B$ 的中值。
- 7.39 设有 $N$ 个元素的数组只包含两个不同的键true和false。给出一个 $O(N)$ 算法，重新排列这些元素，使得所有false的元素都排在true的元素的前面。你只能使用常数附加空间。
- 7.40 设有 $N$ 个元素的数组包含三个不同的键true、false和maybe。给出一个 $O(N)$ 算法，重新排列这些元素，使得所有false的元素都排在maybe元素的前面，而maybe元素都排在true元素的前面。你只能使用常数附加空间。
- 7.41 a. 证明，任何基于比较的算法将4个元素排序均需5次比较。  
b. 给出一个算法，用5次比较将4个元素排序。
- 7.42 a. 证明使用任何基于比较的算法将5个元素排序都需要7次比较。  
\*b. 给出一个算法，用7次比较将5个元素排序。
- 7.43 写出一个高效的谢尔排序算法并比较当使用下列增量序列时的性能：  
a. 谢尔的原始序列。  
b. Hibbard的增量。  
c. Knuth的增量： $h_i = \frac{1}{2}(3^i + 1)$ 。  
d. Gonnet的增量 $h_i = \left\lfloor \frac{N}{2.2} \right\rfloor$ ，而 $h_k = \left\lfloor \frac{h_{k+1}}{2.2} \right\rfloor$ （若 $h_2 = 2$ 则 $h_1 = 1$ ）。  
e. Sedgewick的增量。
- 7.44 实现优化的快速排序算法并用下列组合进行实验：  
a. 枢纽元：第一个元素，中间的元素，随机的元素，三数中值，五数中值。  
b. 截止值从0到20。
- 7.45 编写一个例程，读入两个用字母表示的文件并将它们合并到一起，形成第三个也是用字母表示的文件。
- 7.46 设我们实现三数中值例程如下：找出 $a[\text{left}]$ 、 $a[\text{center}]$ 和 $a[\text{right}]$ 的中值，并将它与 $a[\text{right}]$ 交换。以通常的分割方法进行，开始时 $i$ 在left处且 $j$ 在right-1处（而不是left+1和right-2）。  
a. 设输入为2, 3, 4, ...,  $N-1$ ,  $N$ , 1。对于该输入，这种快速排序算法的运行时间是多少？  
b. 设输入数据呈反序排列，对于该输入，这种快速排序算法的运行时间又是多少？
- 7.47 证明，任何基于比较的排序算法都平均需要 $\Omega(M \log N)$ 次比较。
- 7.48 给定一个数组，该数组包含 $N$ 个元素。我们想要确定是否存在两个数，它们的和等于给定的数 $K$ 。例如，如果输入是8, 4, 1, 6而 $K$ 是10，则答案为yes（4和6）。一个数可以被使用两次。解答下列各问：  
a. 给出求解该问题的 $O(N^2)$ 算法。  
b. 给出求解该问题的 $O(M \log N)$ 算法（提示：首先将各项排序。然后，可以以线性时间解决该问题）。  
c. 将两种方案编码并比较算法的运行时间。
- 7.49 对于4个数重复练习7.48。尝试设计一个 $O(N^2 \log N)$ 算法（提示：计算两个元素的所有可能的和，把这些可能的和排序，然后按练习7.48来处理）。
- 7.50 对于3个数重复练习7.48。尝试设计一个 $O(N^2)$ 算法。
- 7.51 考虑下面的percolateDown算法。我们在结点 $X$ 有一个空穴（hole）。普通的例程是比较 $X$ 的儿子然后把比我们企图要放置的元素大的儿子上移到 $X$ 处（在（max）堆的情形下），由此将空穴下推；当把新元素安全放到空穴中时终止算法。另一种方法是将一些元素上移且空穴尽可能地下移，不用测试是否能够往新单元插入。这将使得新单元被放置到一片树叶上并可能破坏堆序性质；为了修复堆序，以通常的方式将新单元上滤。写出包含该想法的例程，并与标准的堆排序实现方法的运行时间进行比较。

- 7.52 提出一种算法, 只用两盘磁带对一个大型文件进行排序。
- 7.53 a. 通过buildHeap最多使用 $2N$ 次比较的事实, 证明堆个数的下界 $N!/2^{2N}$ 。  
b. 利用Stirling公式扩展该界。
- 7.54 对图7-20中的Pointer类, 是否需要使用零参数构造函数?
- 7.55 通过重载其成员名为operator\*的函数来检查NULL指针, 使智能指针类Pointer更智能。当试图解引用一个NULL指针时, 输出错误信息, 否则返回\*pointee。
- 7.56 7.8节论述的中间置换分析需要给出每个长度为 $L$ 的平均循环次数。和以前一样, 令 $N$ 为需要排序的元素的个数,  $p$ 为任意位置。  
a. 说明 $p$ 在长度为1的循环中的概率为 $1/N$ 。  
b. 说明 $p$ 在长度为2的循环中的概率为 $1/N$ 。  
c. 说明 $p$ 在长度为 $L$ 的循环中的概率为 $1/N$ 。  
d. 基于(c)推导长度为 $L$ 的循环次数为 $1/L$  (提示: 每个元素对长度 $L$ 的循环次数的贡献为 $1/N$ , 但是简单的累加会过多计算循环次数)。  
e. 说明Comparable复制的平均数是 $N-2+H_N$ 。

## 参考文献

Knuth的书[12]是一本排序的综合参考文献。Gonnet和Baeza-Yates[5]含有一些新成果以及大量的文献目录。

详细论述谢尔排序的原始论文是[24]。Hibbard的论文[6]提出增量 $2^k-1$ 的使用并通过避免交换紧缩了程序。定理7.4源自[15]。Pratt的下界可以在[17]中找到, 他用到的方法比书中提到的方法要复杂。改进的增量序列和上界出现在论文[10]、[23]和[26]中; 匹配的下界见于[27]。最近的一个结果指出, 没有增量序列能够给出 $O(M \log N)$ 的最坏情形运行时间[16]。谢尔排序的平均情形运行时间仍然没有解决。Yao[29]对3增量情形进行了极其复杂的分析。其结果尚需扩展到更多增量, 不过最近稍微有所改进[11]。对各种增量序列的试验见于论文[25]。

堆排序由Williams发现[28]; Floyd[2]提供了构建堆的线性时间算法。定理7.5取自[18]。

归并排序的精确的平均情形分析在[4]中描述。不用附加空间且以线性时间执行合并的算法在[9]中讨论。

快速排序源于Hoare[7]。这篇论文分析了基本算法, 描述了大部分改进方法, 并且还包含选择算法。详细的分析和经验性的研究曾是Sedgewick的专题论文[22]的主题。许多重要的结果出现在三篇论文[19]、[20]和[21]中。[1]提供了详细的C实现并包括某些改进, 它还指出UNIX大部分qsort库函数的实现方法容易导致二次的特性。练习7.27取自[14]。

决策树和排序优化在Ford和Johnson[3]中讨论。这篇论文还提供了一个算法, 它几乎符合用比较 (而不是其他操作) 次数表示的下界。该算法最终由Manacher[13]指出稍逊于最优。

外部排序及其细节涵盖于[12]。练习7.31中描述的稳定排序算法已由Horvath[8]提出。

1. J. L. Bentley and M. D. McElroy, "Engineering a Sort Function," *Software—Practice and Experience*, 23 (1993), 1249-1265.
2. R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, 7 (1964), 701.
3. L. R. Ford and S. M. Johnson, "A Tournament Problem," *American Mathematics Monthly*, 66(1959), 387-389.
4. M. Golin and R. Sedgewick, "Exact Analysis of Mergesort," *Fourth SIAM Conference on Discrete Mathematics*, 1988.
5. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2nd ed., Addison-Wesley, Reading, Mass., 1991.
6. T. H. Hibbard, "An Empirical Study of Minimal Storage Sorting," *Communications of the ACM*, 6

- (1963), 206-213.
7. C. A. R. Hoare, "Quicksort," *Computer Journal*, 5 (1962), 10-15.
  8. E. C. Horvath, "Stable Sorting in Asymptotically Optimal Time and Extra Space," *Journal of the ACM*, 25 (1978), 177-199.
  9. B. Huang and M. Langston, "Practical In-place Merging," *Communications of the ACM*, 31 (1988), 348-352.
  10. J. Incerpi and R. Sedgewick, "Improved Upper Bounds on Shellsort," *Journal of Computer and System Sciences*, 31 (1985), 210-224.
  11. S. Janson and D. E. Knuth, "Shellsort with Three Increments," *Random Structures and Algorithms*, 10 (1997), 125-142.
  12. D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching*, 2d ed., Addison-Wesley, Reading, Mass., 1998.
  13. G. K. Manacher, "The Ford-Johnson Sorting Algorithm Is Not Optimal," *Journal of the ACM*, 26 (1979), 441-456.
  14. D. R. Musser, "Introspective Sorting and Selection Algorithms," *Software—Practice and Experience*, 27 (1997), 983-993.
  15. A. A. Papernov and G. V. Stasevich, "A Method of Information Sorting in Computer Memories," *Problems of Information Transmission*, 1 (1965), 63-75.
  16. C. G. Plaxton, B. Poonen, and T. Suel, "Improved Lower Bounds for Shellsort," Proceedings of the Thirty-third Annual Symposium on the Foundations of Computer Science (1992), 226-235.
  17. V. R. Pratt, *Shellsort and Sorting Networks*, Garland Publishing, New York, 1979. (Originally presented as the author's Ph.D. thesis, Stanford University, 1971.)
  18. R. Schaffer and R. Sedgewick, "The Analysis of Heapsort," *Journal of Algorithms*, 14 (1993), 76-100.
  19. R. Sedgewick, "Quicksort with Equal Keys," *SIAM Journal on Computing*, 6 (1977), 240-267.
  20. R. Sedgewick, "The Analysis of Quicksort Programs," *Acta Informatica*, 7 (1977), 327-355.
  21. R. Sedgewick, "Implementing Quicksort Programs," *Communications of the ACM*, 21 (1978), 847-857.
  22. R. Sedgewick, *Quicksort*, Garland Publishing, New York, 1978. (Originally presented as the author's Ph.D. thesis, Stanford University, 1975.)
  23. R. Sedgewick, "A New Upper Bound for Shellsort," *Journal of Algorithms*, 7 (1986), 159-173.
  24. D. L. Shell, "A High-Speed Sorting Procedure," *Communications of the ACM*, 2 (1959), 30-32.
  25. M. A. Weiss, "Empirical Results on the Running Time of Shellsort," *Computer Journal*, 34 (1991), 88-91.
  26. M. A. Weiss and R. Sedgewick, "More on Shellsort Increment Sequences," *Information Processing Letters*, 34 (1990), 267-270.
  27. M. A. Weiss and R. Sedgewick, "Tight Lower Bounds for Shellsort," *Journal of Algorithms*, 11 (1990), 242-251.
  28. J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, 7 (1964), 347-348.
  29. A. C. Yao, "An Analysis of  $(h, k, 1)$  Shellsort," *Journal of Algorithms*, 1 (1980), 14-50.

# 不相交集类

在这一章，我们描述解决等价问题的一种有效数据结构。这种数据结构实现起来很简单，**在**每个例程只需要几行代码，而且可以使用一个简单的数组。它的实现也非常快，每种操作只需要常数平均时间。从理论上讲，这种数据结构还是非常有趣的，因为它的分析极其困难；最坏情形的函数形式不同于我们已经见过的任何形式。对于这种不相交集数据结构，我们将：

- 讨论如何能够以最少的编程代价实现。
- 通过两个简单的观察结果极大地提高它的速度。
- 分析一种快速的实现方法的运行时间。
- 介绍一个简单的应用。

## 8.1 等价关系

若对于每一对元素 $(a, b)$ ， $a, b \in S$ ， $a R b$ 或者为true或者为false，则称在集合 $S$ 上定义关系（relation） $R$ 。如果 $a R b$ 为true，那么我们说 $a$ 与 $b$ 有关系。

等价关系（equivalence relation）是满足下列三个性质的关系 $R$ ：

- (1) 自反性：对于所有的 $a \in S$ ， $a R a$ 。
- (2) 对称性： $a R b$ 当且仅当 $b R a$ 。
- (3) 传递性：若 $a R b$ 且 $b R c$ 则 $a R c$ 。

下面考虑几个例子。

关系“ $\leq$ ”不是等价关系。虽然它是自反的（即 $a \leq a$ ）、可传递的（即由 $a \leq b$ 和 $b \leq c$ 得出 $a \leq c$ ），但它却不是对称的，因为从 $a \leq b$ 并不能得出 $b \leq a$ 。

电气连通性（electrical connectivity）是一个等价关系，其中所有的连接都是通过金属导线完成的。该关系显然是自反的，因为任何元件都是自身相连的。如果 $a$ 电气连接到 $b$ ，那么 $b$ 必然也电气连接到 $a$ ，因此关系是对称的。最后，如果 $a$ 连接到 $b$ ，而 $b$ 又连接到 $c$ ，那么 $a$ 连接到 $c$ 。因此，电气连接是一个等价关系。

如果两个城市位于同一个国家，那么定义它们是有关系的。容易验证这是一个等价关系。如果能够通过公路从城镇 $a$ 旅行到 $b$ ，则设 $a$ 与 $b$ 有关系。如果所有的道路都是双向行驶的，那么这种关系也是一个等价关系。

## 8.2 动态等价性问题

给定一个等价关系“ $\sim$ ”，一个自然的问题是对任意的 $a$ 和 $b$ ，确定是否 $a \sim b$ 。如果将等价关系存储为一个二维布尔数组，那么当然这个工作可以以常数时间完成。问题在于，关系的定义通常



不明显而且相当隐秘。

作为一个例子，设在5个元素的集合 $\{a_1, a_2, a_3, a_4, a_5\}$ 上定义一个等价关系。此时存在25对元素，其中每一对或者有关系或者没有关系。然而，信息 $a_1 \sim a_2, a_3 \sim a_4, a_5 \sim a_1, a_4 \sim a_2$ 意味着每一对元素都是有关系的。我们希望能够迅速推断出这些关系。

元素 $a \in S$ 的**等价类** (equivalence class) 是 $S$ 的子集，它包含所有与 $a$ 有(等价)关系的元素。注意，等价类形成对 $S$ 的一个划分： $S$ 的每一个成员恰好出现在一个等价类中。为确定是否 $a \sim b$ ，我们只需验证 $a$ 和 $b$ 是否都在同一个等价类中。这给我们提供了解决等价问题的方法。

输入数据最初是 $N$ 个集合的类(collection)，每个集合含有一个元素。初始的描述是所有的关系均为false(自反的关系除外)。每个集合都有一个不同的元素，从而 $S_i \cap S_j = \emptyset$ ；这使得这些集合**不相交** (disjoint)。

此时，有两种操作允许进行。第一种操作是find，它返回包含给定元素的集合(即等价类)的名字。第二种操作是添加关系。如果我们想要添加关系 $a \sim b$ ，那么首先要看是否 $a$ 和 $b$ 已经有关系。这可以通过对 $a$ 和 $b$ 执行find并检验它们是否在同一个等价类中来完成。如果它们不在同一类中，那么使用求并操作union<sup>1</sup>，这种操作把含有 $a$ 和 $b$ 的两个等价类合并成一个新的等价类。从集合的观点来看，“ $\cup$ ”的结果是建立一个新集合 $S_k = S_i \cup S_j$ ，去掉原来两个集合而保持所有的集合的不相交性。由于这个原因，常常把做这项工作的算法叫作不相交集类的**求并/查找** (union/find) 算法。

该算法是动态的，因为在算法执行的过程中，集合可以通过union操作而发生改变。这个算法还必然是**联机** (on-line) 操作：当find执行时，它必须给出答案算法才能继续进行。另一种可能是**脱机** (off-line) 算法，该算法需要观察全部的union和find序列。对每个find它给出的答案必须和所有执行到该find的union一致，不过该算法在看到所有这些问题以后再给出它的所有的答案。这种差别类似于参加一次笔试(它一般是脱机的——只能在规定的时间内用完之前给出答卷)和一次口试(它是联机的，因为你必须回答当前的问题，然后才能继续下一个问题)。

316

注意，我们不进行任何比较元素相关的值的操作，而是只需要知道它们的位置。由于这个原因，假设所有的元素均已从0到 $N-1$ 顺序编号并且编号方法容易由某个散列方案确定。于是，开始时我们有 $S_i = \{i\}, i = 0$ 到 $N-1$ 。<sup>2</sup>

我们的第二个观察结果是，由find返回的集合的名字实际上是相当任意的。真正重要的关键点在于： $\text{find}(a) == \text{find}(b)$  为true，当且仅当 $a$ 和 $b$ 在同一个集合中。

这些操作在许多图论问题中是重要的，在一些处理等价(或类型)声明的编译程序中也很重要。我们将在后面讨论一个应用。

解决动态等价问题的方案有两种。一种方案保证指令find能够以常数最坏情形运行时间执行，而另一种方案则保证指令union能够以常数最坏情形运行时间执行。已经证明二者不能同时以常数最坏情形运行时间执行。

我们将简要讨论第一种处理方法。为使find操作快，可以在一个数组中保存每个元素的等价类的名字。此时，find就是简单的 $O(1)$ 查找。设我们想要执行 $\text{union}(a, b)$ ，并设 $a$ 在等价类 $i$ 中而 $b$ 在等价类 $j$ 中。此时我们扫描该数组，将所有的 $i$ 都改变成 $j$ 。不过，这次扫描要花费 $\Theta(N)$ 时间。于是，连续 $N-1$ 次union操作(这是最大值，因为此时每个元素都在一个集合中)就要花费 $\Theta(N^2)$

1. Union 是(很少使用的) C++ 的一个保留字。在整个描述 union/find 算法的过程中，我们使用这个字，但是当编写代码时成员函数将命名为 unionSets。

2. 这反映了数组下标从 0 开始的事实。

的时间。如果存在 $\Omega(N^2)$ 次find操作,那么性能会很好,因为在整个算法进行过程中每个union或find操作的总的运行时间为 $O(1)$ 。如果find操作没有那么多,那么这个界是不可接受的。

一种想法是将所有在同一个等价类中的元素放到一个链表中。这在更新的时候会节省时间,因为我们不必搜索整个数组。但是由于在算法过程中仍然可能执行 $\Theta(N^2)$ 次等价类的更新,因此它本身并不能单独减少渐近运行时间。

如果我们还要跟踪每个等价类的大小,并在执行union时将较小的等价类的名字改成较大的等价类的名字,那么对于 $N-1$ 次合并的总的时间开销为 $O(M \log N)$ 。其原因在于,每个元素可能将它的等价类最多改变 $\log N$ 次,因为每次等价类改变时它的新的等价类至少是其原来等价类的两倍大。使用这种方法,任意顺序的 $M$ 次find和直到 $N-1$ 次的union最多花费 $O(M + M \log N)$ 时间。

在本章的其余部分,我们将考察union/find问题的一种解法,其中union操作容易但find操作要难一些。即使如此,任意顺序的最多 $M$ 次find和直到 $N-1$ 次union的运行时间将只比 $O(M + N)$ 多一点。

## 8.3 基本数据结构

回想以下问题:不要求find操作返回任何特定的名字,而只要求当且仅当两个元素属于相同的集合时,作用在这两个元素上的find返回相同的名字。一种想法是可以使用树来表示每一个集合,因为树上的每一个元素都有相同的根。这样,该根就可以用来命名所在的集合。我们将用树表示每一个集合(我们知道,树的集合叫作森林)。开始时每个集合含有一个元素。这些树不一定必须是二叉树,但是用二叉树表示起来要容易,因为我们需要的唯一信息就是父链(parent link)。集合的名字由根处的结点给出。由于只需要父结点的名字,因此可以假设这棵树被非显式地存储在数组中:数组的每个成员 $s[i]$ 表示元素 $i$ 的父亲。如果 $i$ 是根,那么 $s[i] = -1$ 。在图8-1的森林中,对于 $0 \leq i \leq 8$ , $s[i] = -1$ 。正如二叉堆中那样,我们也将显式地画出这些树,注意,此时正在使用一个数组。图8-1表达了这种显式的表示方法,为方便起见,把根的父链垂直画出。

317

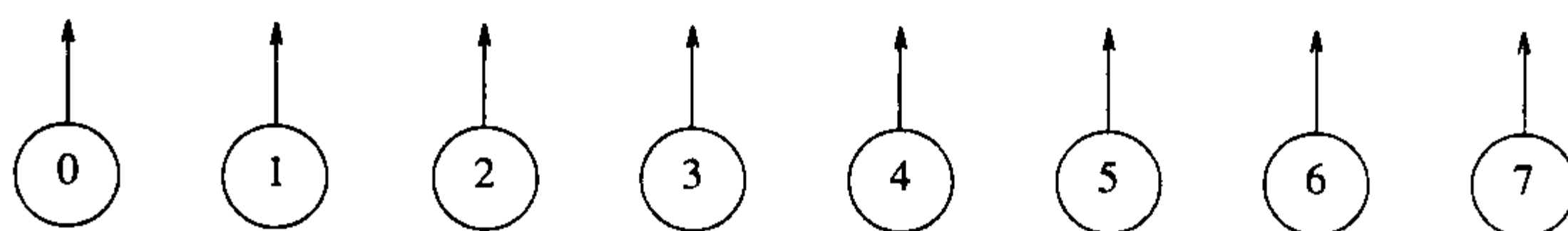


图8-1 8个元素,初始时在不同的集合上

为了执行两个集合的union操作,我们通过使一棵树的根的父链链接到另一棵树的根结点合并两棵树。显然,这种操作花费常数时间。图8-2、图8-3和图8-4分别表示在 $\text{union}(4,5)$ 、 $\text{union}(6,7)$ 和 $\text{union}(4,6)$ 操作之后的森林,其中,我们采纳了在 $\text{union}(x,y)$ 后新的根是 $x$ 的约定。最后的森林的非显式表示见图8-5。

318

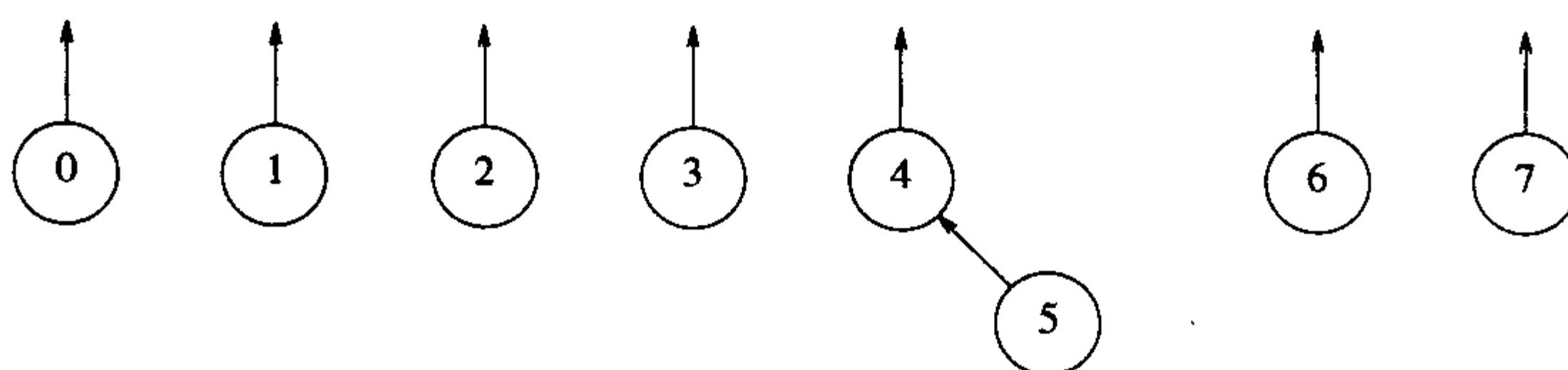


图8-2 在 $\text{union}(4,5)$ 之后

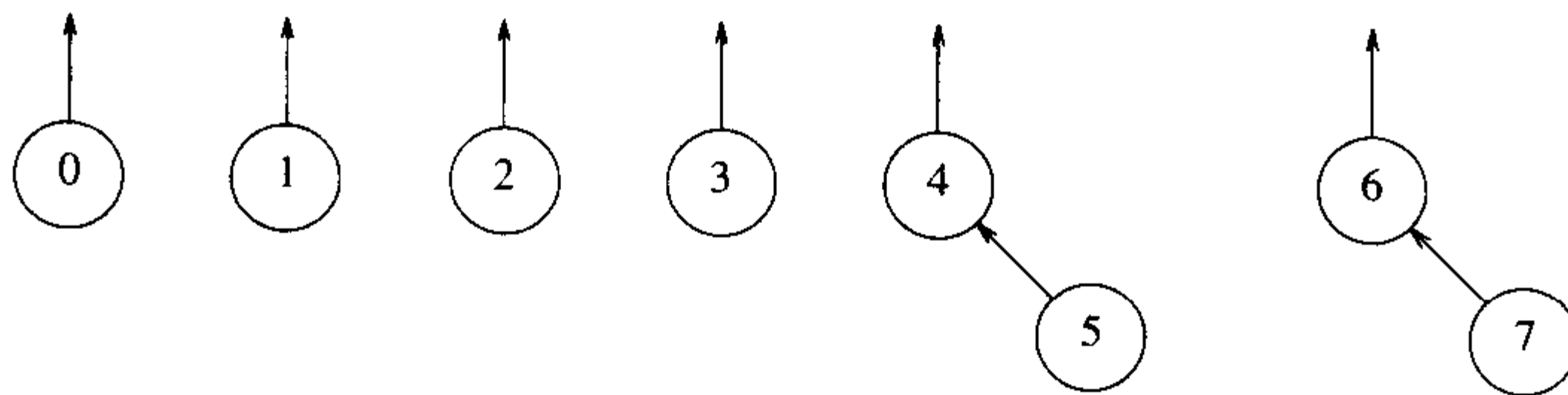


图8-3 在union(6,7)之后

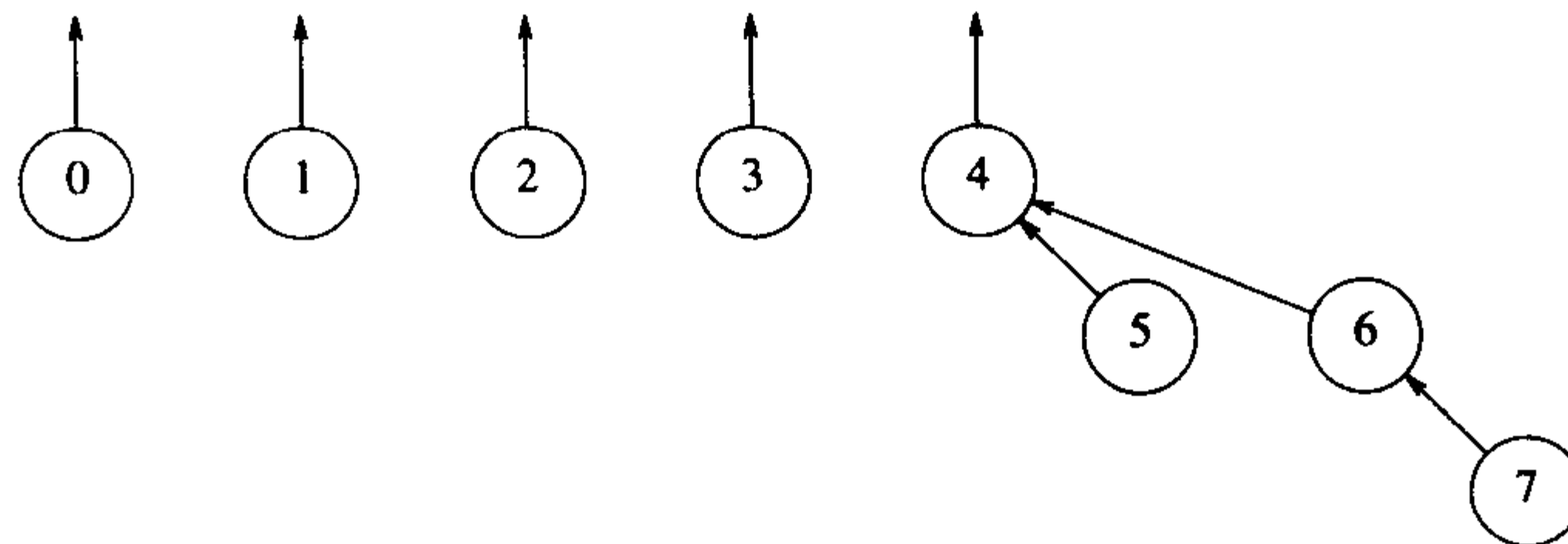


图8-4 在union(4,6)之后

|    |    |    |    |    |   |   |   |
|----|----|----|----|----|---|---|---|
| -1 | -1 | -1 | -1 | -1 | 4 | 4 | 6 |
| 0  | 1  | 2  | 3  | 4  | 5 | 6 | 7 |

图8-5 上面的树的非显式表示

对元素 $x$ 的一次 $\text{find}(x)$ 操作通过返回包含 $x$ 的树的根而完成。执行这次操作花费的时间与表示 $x$ 的结点的深度成正比，当然这要假设我们以常数时间找到表示 $x$ 的结点。使用上面的方法，能够建立一个深度为 $N-1$ 的树，因此 $\text{find}$ 的最坏情形运行时间是 $O(N)$ 。一般情况下，运行时间是针对连续混合使用 $M$ 个指令来计算的。在这种情况下， $M$ 次连续操作在最坏情形下可能花费 $O(MN)$ 时间。

**319** 图8-6到图8-9中的程序表示基本算法的实现，假设差错检验已经执行。在我们的例程中， $\text{union}$ 是在树的根上进行的。有时候操作是通过传递任意两个元素进行，并使得 $\text{union}$ 执行两次 $\text{find}$ 以确定根。

```

1 class DisjSets
2 {
3     public:
4         explicit DisjSets( int numElements );
5
6         int find( int x ) const;
7         int find( int x );
8         void unionSets( int root1, int root2 );
9
10    private:
11        vector<int> s;
12 };

```

图8-6 不相交集的类架构

```

1 /**
2  * Construct the disjoint sets object.
3  * numElements is the initial number of disjoint sets.
4  */
5 DisjSets::DisjSets( int numElements ) : s( numElements )
6 {
7     for( int i = 0; i < s.size( ); i++ )
8         s[ i ] = -1;
9 }

```

图8-7 不相交集的初始化例程

```

1  /**
2   * Union two disjoint sets.
3   * For simplicity, we assume root1 and root2 are distinct
4   * and represent set names.
5   * root1 is the root of set 1.
6   * root2 is the root of set 2.
7   */
8  void DisjSets::unionSets( int root1, int root2 )
9  {
10     s[ root2 ] = root1;
11 }

```

图8-8 union (不是最好的方法)

```

1  /**
2   * Perform a find.
3   * Error checks omitted again for simplicity.
4   * Return the set containing x.
5   */
6  int DisjSets::find( int x ) const
7  {
8     if( s[ x ] < 0 )
9         return x;
10    else
11        return find( s[ x ] );
12 }

```

图8-9 一个简单的不相交集find算法

平均情形分析是相当困难的。最起码的问题是答案依赖于如何定义（对union操作而言的）平均。例如，在图8-4的森林中，可以说，由于有5棵树，因此下一次union就存在 $5 \times 4 = 20$ 个等可能的结果（因为任意两棵不同的树都可能被union）。当然，这个模型的含义在于，只存在 $2/5$ 的机会使得下一次union涉及大树。另一种模型可能会认为在不同的树上任意两个元素间的所有union都是等可能的，因此大树比小树更有可能在下一次union中涉及。在上面的例子中，有 $8/11$ 的机会大树在下一次union中被涉及，因为（忽略对称性）存在6种方法合并 $\{0, 1, 2, 3\}$ 中的两个元素以及16种方法将 $\{4, 5, 6, 7\}$ 中的一个元素与 $\{0, 1, 2, 3\}$ 中的一个元素合并。还存在更多的模型，而在哪个为最好的问题上没有一般的一致见解。平均运行时间依赖于模型；对于三种不同的模型，时间界 $\Theta(M)$ 、 $\Theta(M \log N)$ 以及 $\Theta(MN)$ 实际上已经证明，不过，最后的那个界更现实些。

对一系列操作的二次运行时间一般是不可接受的。幸运的是，有几种方法容易保证这样的运行时间不会出现。

## 8.4 灵巧求并算法

上面的union的执行是相当随意的，它通过使第二棵树成为第一棵树的子树而完成合并。对其进行简单改进是借助任意的方法打破现有的随意性，使得总是较小的树成为较大的树的子树；我们把这种方法叫作按大小求并（union by size）。前面的例子中三次union的对象大小都是一样的，因此可以认为它们都是按照大小执行的。假如下一次操作是union(3, 4)，那么结果将形成图8-10中的森林。倘若没有对大小进行探测而直接union，那么结果将会形成更深的树（见图8-11）。



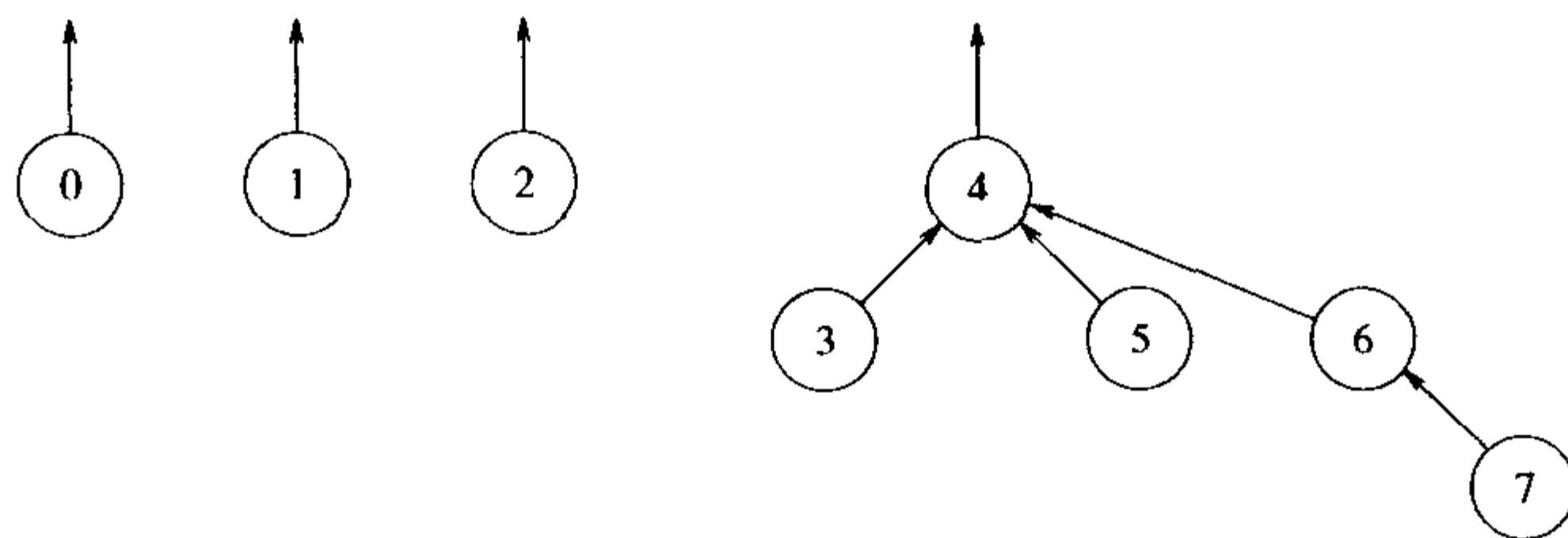


图8-10 按大小求并的结果

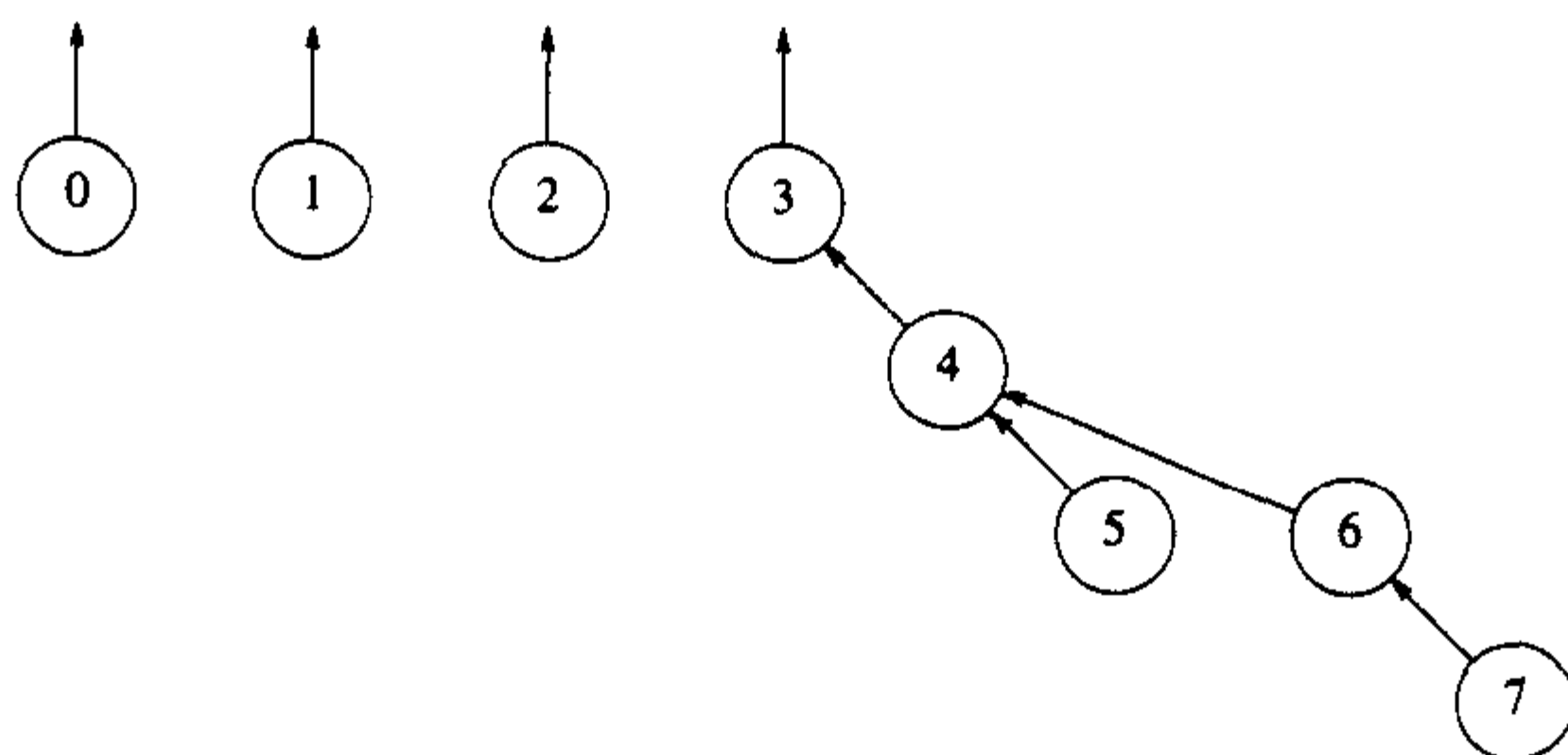
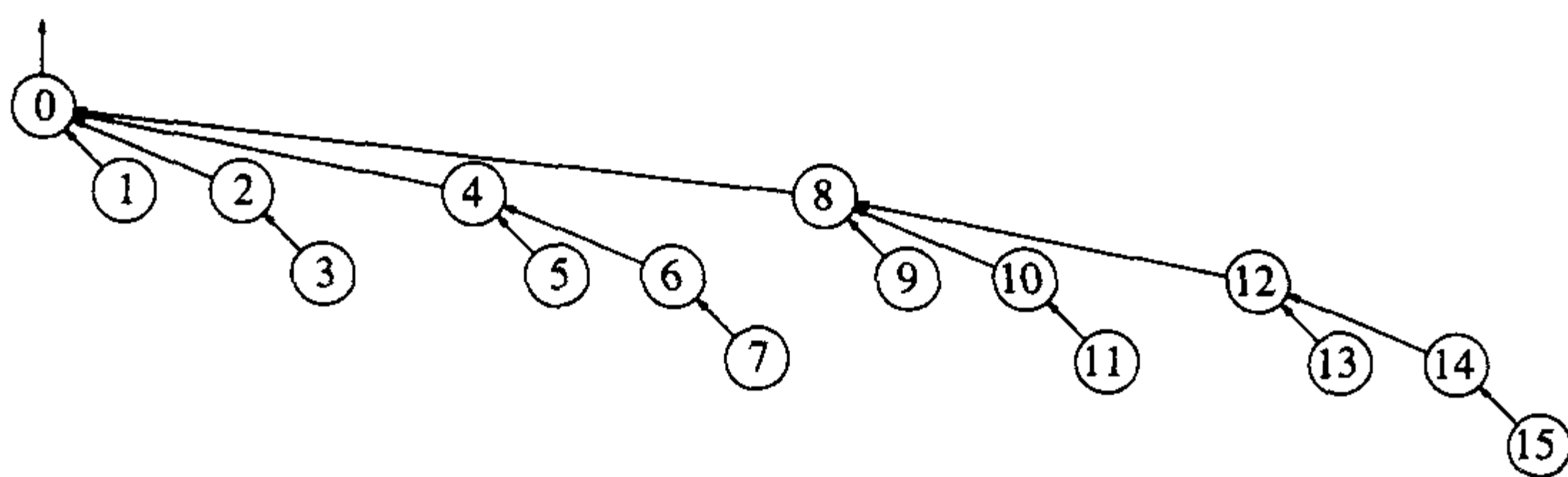


图8-11 进行一次任意的union的结果

可以证明，如果这些union都是按照大小进行的，那么任何结点的深度均不会超过 $\log N$ 。为此，首先注意结点初始处于深度0的位置。当它的深度随着一次union的结果而增加的时候，该结点则被置于至少是它以前所在树两倍大的一棵树上。因此，它的深度最多可以增加 $\log N$ 次（我们在8.2节末的快速查找算法中用过这个论断）。这意味着，find操作的运行时间是 $O(\log N)$ ，而连续 $M$ 次操作则花费 $O(M \log N)$ 。图8-12中的树指出在16次union后有可能得到这种最坏情形的树，而且如果所有的union都对相等大小的树进行，那么这样的树是可能得到的（最坏情形的树是在第6章讨论过的二项树）。

图8-12  $N = 16$ 时最坏情形的树

为了实现这种方法，我们需要记住每一棵树的大小。由于实际上只使用一个数组，因此可以让每个根的数组元素包含它的树的大小的负值。这样一来，初始时树的数组表示就都是-1了。当执行一次union时，要检查树的大小；新的大小是原来的大小的和。这样，按大小求并的实现根本不存在困难，并且不需要额外的空间，其速度平均也很快。对于所有合理的模型，业已证明，若使用按大小求并则连续 $M$ 次操作平均需要 $O(M)$ 时间。这是因为当随机的union执行时整个算法一般只有一些很小的集合（通常含一个元素）与大集合合并。

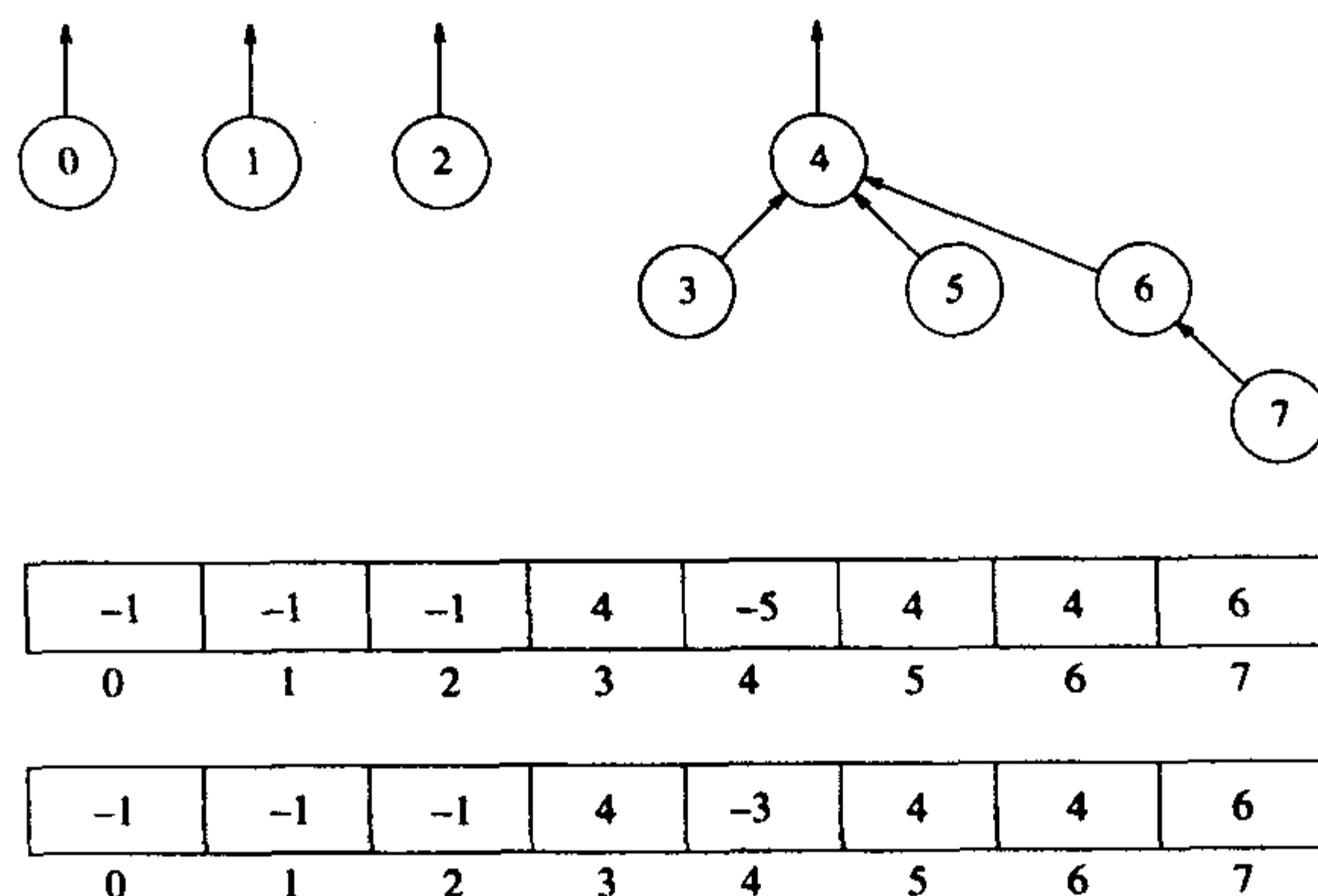
322

另外一种实现方法为按高度求并（union-by-height），它同样保证所有的树的深度最多是 $O(\log N)$ 。我们跟踪每棵树的高度而不是大小并执行union使得浅的树成为深的树的子树。这是一种平缓的算法，因为只有当两棵相等深度的树求并时树的高度才增加（此时树的高度增1）。因此，按高度求并是按大小求并的简单修改。由于零的高度不是负的，因此实际上存储高度的负值，减

去附加的1。初始时所有的项都是-1。

下列各图显示了一棵树及其对于按大小求并和按高度求并的非显式表示。图8-13中的程序实现的是按高度求并。

323



```

1 /**
2  * Union two disjoint sets.
3  * For simplicity, we assume root1 and root2 are distinct
4  * and represent set names.
5  * root1 is the root of set 1.
6  * root2 is the root of set 2.
7  */
8 void DisjSets::unionSets( int root1, int root2 )
9 {
10     if( s[ root2 ] < s[ root1 ] ) // root2 is deeper
11         s[ root1 ] = root2;      // Make root2 new root
12     else
13     {
14         if( s[ root1 ] == s[ root2 ] )
15             s[ root1 ]--;        // Update height if same
16         s[ root2 ] = root1;      // Make root1 new root
17     }
18 }

```

图8-13 按高度（秩）求并的程序

## 8.5 路径压缩

迄今所描述的union/find算法对于大多数的情形都是完全可接受的，它是非常简单的，而且对于连续 $M$ 个指令（在所有的模型下）平均是线性的。不过， $O(M \log N)$ 的最坏情形还是相当容易和自然地发生的。例如，如果我们把所有的集合放到一个队列中并重复地让前两个集合出队而让它们的并入队，那么最坏情形就会发生。如果find比union多很多，那么其运行时间就比快速查找算法（quick-find algorithm）的用时要多。而且应该清楚，对于union算法恐怕没有更多改进的可能。这是基于这样的观察：执行union操作的任何算法都将产生相同的最坏情形的树，因为它必然会任意打破树间的均衡。因此，无需对整个数据结构重新加工而使算法加速的唯一方法是对find操作做些更巧妙的改进。

这种巧妙的改进叫作**路径压缩**（path compression）。路径压缩在一次find操作期间执行而与用来执行union的方法无关。设操作为find( $x$ )，此时路径压缩的效果是，从 $x$ 到根的路径上的每

一个结点都使它的父结点变成根。图8-14给出对图8-12的最坏的树执行find(14)后路径压缩的效果。

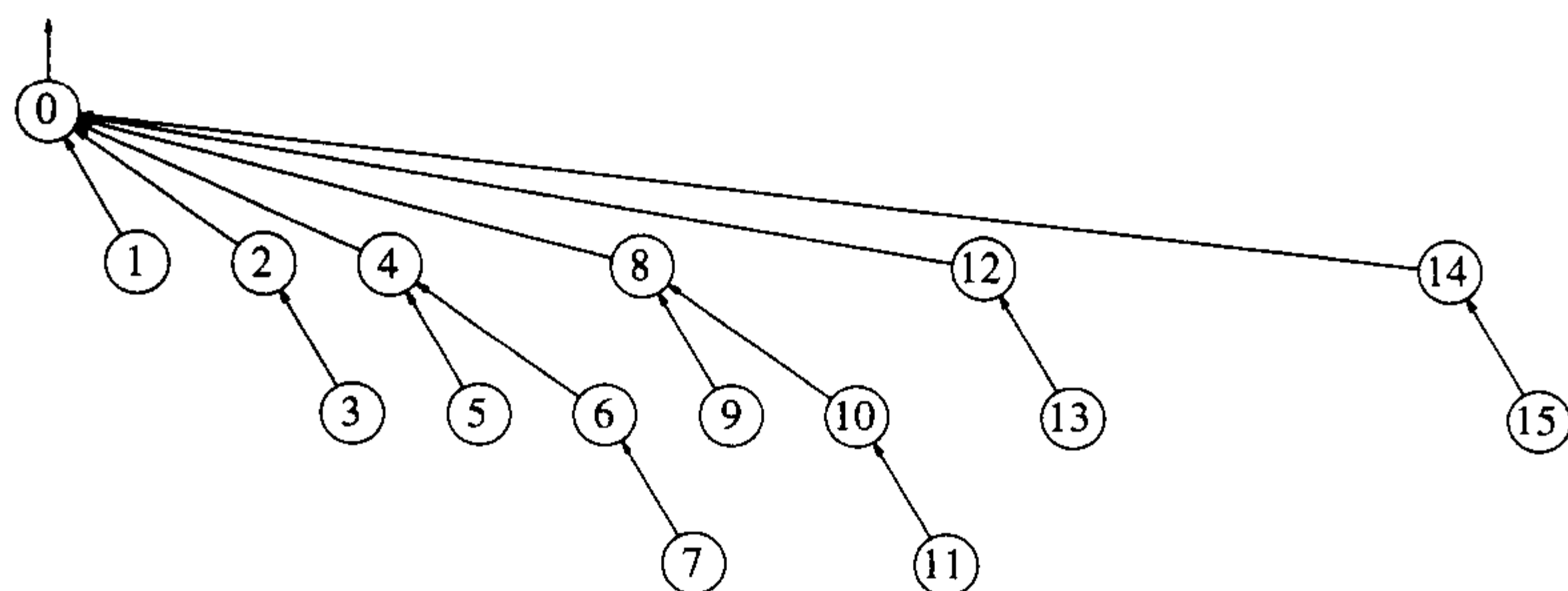


图8-14 路径压缩的一个例子

路径压缩的效果在于通过额外的两个链的变化，结点12和13现在离根近了一个位置，而结点14和15现在离根近了两个位置。因此，以后对这些结点的访问将（我们希望）由于花费额外的工作来进行路径压缩而得到补偿。

正如图8-15中的程序所指出的，路径压缩对基本的find算法的改变不大。对find例程来说，唯一的变化是使得s[x]等于由find返回的值；这样，在集合的根被递归地找到以后，x的父链就引用它。这对通向根的路径上的每一个结点递归地出现，因此实现了路径压缩。

```

1  /**
2   * Perform a find with path compression.
3   * Error checks omitted again for simplicity.
4   * Return the set containing x.
5   */
6  int DisjSets::find( int x )
7  {
8      if( s[ x ] < 0 )
9          return x;
10     else
11         return s[ x ] = find( s[ x ] );
12 }

```

图8-15 用路径压缩进行不相交集find的程序

当任意执行union操作的时候，路径压缩是一个好的想法，因为存在许多的深层结点，可通过路径压缩将它们移近根结点。业已证明，当在这种情况下进行路径压缩时，连续 $M$ 次操作最多需要 $O(M \log N)$ 的时间。不过，在这种情况下确定平均情形的性能如何仍然是一个尚未解决的问题。

路径压缩与按大小求并完全兼容，这就使得两个例程可以同时实现。由于按大小求并本身要以线性时间执行连续 $M$ 次操作，因此还不清楚在路径压缩中涉及的额外一趟工作平均来讲是否值得。这个问题实际上仍然没有解决。不过后面我们将会看到，路径压缩与灵巧求并法则结合在所有情况下都将产生非常有效的算法。

路径压缩不完全与按高度求并兼容，因为路径压缩可以改变树的高度。我们根本不清楚如何有效地去重新计算它们。答案是不计算!! 此时，对于每棵树所存储的高度是估计的高度（有时称为秩），但按秩求并（正是现在已经变成的样子）理论上和按大小求并效率是一样的。不仅如此，高度的更新也不如大小的更新频繁。与按大小求并一样，我们也不清楚路径压缩平均来讲是否值得。下一节将证明，使用求并探测法或路径压缩都能够显著地减少最坏情形运行时间。

## 8.6 按秩求并和路径压缩的最坏情形

当使用两种探测法时，算法在最坏情形下几乎是线性的。特别地，在最坏情形下需要的时间是 $\Theta(M\alpha(M, N))$ （假设 $M \geq N$ ），其中， $\alpha(M, N)$ 是Ackermann函数的逆，Ackermann函数如下定义<sup>1</sup>： 325

$$\begin{aligned} A(1, j) &= 2^j, j \geq 1 \\ A(i, 1) &= A(i-1, 2), i \geq 2 \\ A(i, j) &= A(i-1, A(i, j-1)), i, j \geq 2 \end{aligned}$$

由此，定义

$$\alpha(M, N) = \min\{i \geq 1 \mid A(i, \lfloor M/N \rfloor) > \log N\}$$

你可能想要计算某些值，不过在实际应用中 $\alpha(M, N) \leq 4$ ，对我们才是真正重要的。单变量反Ackermann函数有时写成 $\log^* N$ ，它是 $N$ 的直到 $N \leq 1$ 的取对数的次数。于是， $\log^* 65\,536 = 4$ ，这是因为 $\log \log \log \log 65\,536 = 1$ 。 $\log^* 2^{65\,536} = 5$ ，不过要知道， $2^{65\,536}$ 可是一个20 000位数字的大数。 $\alpha(M, N)$ 实际上甚至比 $\log^* N$ 增长得还慢。然而， $\alpha(M, N)$ 却不是常数，因此运行时间并不是线性的。

在本节的其余部分我们将证明一个稍微弱一些的结果。我们将证明，任意顺序的 $M = \Omega(N)$ 次union/find操作花费的总运行时间为 $O(M \log^* N)$ 。如果用按大小求并代替按秩求并，则这个界同样是成立的。这大概是本书最为复杂的分析工作，也是曾对事实上实现起来非常简单的一个算法进行的第一批真正复杂的最坏情形分析之一。

### 求并/查找算法的分析

在这一小节，我们对连续 $M = \Omega(N)$ 次求并和查找操作的运行时间建立一个相当严格的界，union和find可以以任何顺序出现，但是union是按秩进行而find则利用路径压缩完成。

我们通过建立某些涉及秩 $r$ 的结点个数的引理开始。直观地看，由于按秩求并的法则，小秩的结点要比大秩的结点多得多。特别是，最多存在一个秩为 $\log N$ 的结点。我们想要得出对任意给定的秩 $r$ 的结点个数的一个尽可能精确的界。由于秩仅当union执行（从而仅当两棵树具有相同的秩）时变化，因此可以通过忽略路径压缩来证明这个界。

**引理8.1** 当执行一系列union指令时，一个秩为 $r$ 的结点必然至少有 $2^r$ 个后裔结点（包括它自己）。

**证明** 使用数学归纳法。对于基准情形 $r = 0$ ，引理显然成立。令 $T$ 是秩为 $r$ 的具有最少后裔数的树，并令 $X$ 是 $T$ 的根。设涉及 $X$ 的最后一次union是在 $T_1$ 和 $T_2$ 之间进行的。设 $T_1$ 的根为 $X$ 。如果 $T_1$ 的秩是 $r$ ，那么 $T_1$ 就是一棵高度为 $r$ 的树且比 $T$ 有更少的后裔，这与 $T$ 是具有最少后裔数的树的假设相矛盾。因此 $T_1$ 的秩小于等于 $r - 1$ 。 $T_2$ 的秩小于等于 $T_1$ 的秩。由于 $T$ 有秩 $r$ 而秩只能因 $T_2$ 增加，因此 $T_2$ 的秩为 $r - 1$ 。于是 $T_1$ 的秩为 $r - 1$ 。根据归纳假设，每棵树至少有 $2^{r-1}$ 个后裔，从而总数为 $2^r$ 个后裔，引理得证。 ■

引理8.1告诉我们，如果不进行路径压缩，那么秩为 $r$ 的任意结点必然至少有 $2^r$ 个后裔。当然，326路径压缩可以改变这种状况，因为它能够把后裔从结点上除去。不过，当进行union甚至用到路径压缩时，我们都是在使用秩，这些秩是高度的估计值。这些秩的行为就像是没有路径压缩一样。因此，当确定秩为 $r$ 的结点个数的界时，路径压缩可以忽略。

1. Ackermann 函数的常见形式是下式定义的：当 $j \geq 1$ 时， $A(1, j) = j + 1$ 。本文中的形式增长很快，相应地其逆增长很慢。



于是，下面的定理对于是否有路径压缩都是成立的。

**引理8.2** 秩为 $r$ 的结点的个数最多是 $N/2^r$ 。

**证明** 若无路径压缩，每个秩为 $r$ 的结点都是至少有 $2^r$ 个结点的子树的根。在这样的子树中没有其秩为 $r$ 的结点。因此，那些秩为 $r$ 的结点的所有子树都是不相交的。于是，存在至多 $N/2^r$ 个不相交的子树，从而最多有 $N/2^r$ 个秩为 $r$ 的结点。 ■

下一个引理看似显而易见，但在我们的分析中却是至关重要的。

**引理8.3** 在求并/查找算法的任一时刻，从树叶到根的路径上的结点的秩单调增加。

**证明** 如果不存在路径压缩，那么该引理显然成立（见例子）。如果在路径压缩后某个结点 $v$ 是 $w$ 的一个后裔，那么当只考虑union操作时显然 $v$ 必然还是 $w$ 的后裔。因此， $v$ 的秩小于 $w$ 的秩。 ■

下面总结一下这些初级的结果。引理8.2告诉我们多少结点可以赋予秩 $r$ 。因为秩只能通过union赋值，所以引理8.2在union/find算法的任何阶段甚至在路径压缩的中间都是成立的。图8-16指出，随着 $r$ 的增大秩为 $r$ 的结点变少，可是秩为0和1的结点却有很多。

327

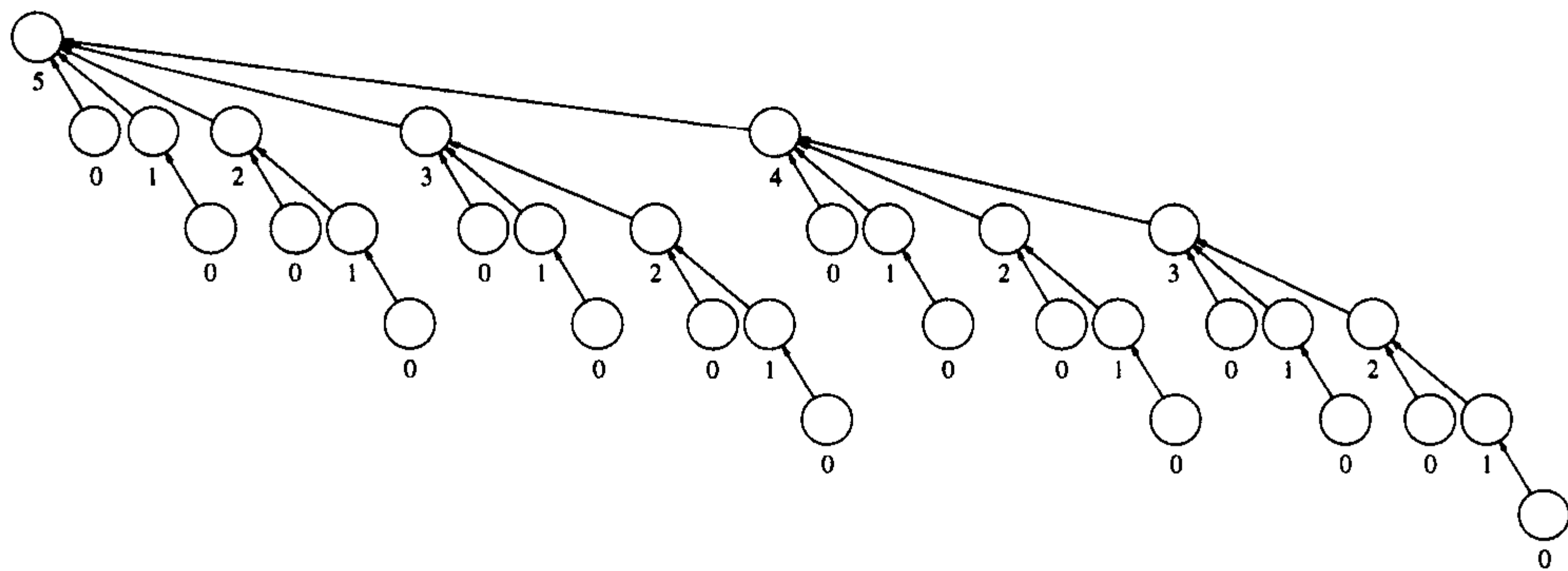


图8-16 一棵大的不相交集树（结点下面的数是秩）

在对任意秩 $r$ 都有可能存在 $N/2^r$ 个结点的意义下，引理8.2是严格的。但该引理还是稍微有些宽松，因为这个界不可能对所有的秩 $r$ 同时成立。引理8.2描述了秩为 $r$ 的结点的个数，而引理8.3则告诉我们它们的分布。正如所期望的，结点的秩沿着从叶到根的路径严格递增。

接下来准备证明主要的定理。证明的基本思想如下：对任何结点 $v$ 的find所花费的时间与从 $v$ 到根的路径上的结点的个数成正比。对每个find在从 $v$ 到根的路径上的每一个结点收取一个单位的费用。为了计算这些费用，想像在路径的每一个结点上存入一美分。严格地说这是一个会计诀窍，它并不是程序的一部分。当算法结束时，将已经存入的所有分币敛起来，这就是总的花费。

作为进一步的会计诀窍，我们存入美分和加拿大分两种分币。我们将证明，在算法执行期间，对于每次find只能存入一定量美分。我们还将证明，只能存入一定量的加拿大分到每一个结点上。把这两笔总数加起来就得到能够存入的分币的总数的界。

现在详细地介绍我们的计算方案。我们将按照秩来划分结点，把秩分成一些秩组。对每个find，将把一些美分币存成公共储金，而把加拿大分币存到一些特定的顶点上<sup>1</sup>。为了计算所存储的加拿大分币的总数，我们将计算每个结点上的储量。通过将秩为 $r$ 的每一个结点的储金加起来，得到每个秩 $r$ 的总的储量。然后，我们再把秩组 $g$ 中每个秩 $r$ 的所有储量加起来从而得到每个

1. 我们互换地使用结点（node）和顶点（vertex）。

秩组 $g$ 的总的储量。最后，把每个秩组 $g$ 的所有储金加到一起就得到在森林中存储的加拿大分币的总数。把这笔储金加到作为公共储金的美分币的数目上则得到最后的答案。

我们将把秩划分成组。秩 $r$ 被分到组 $G(r)$ ，而 $G$ 将在后面确定。任何秩组 $g$ 中最大的秩为 $F(g)$ ，其中 $F = G^{-1}$ 是 $G$ 的逆。于是，在任何秩组 $g > 0$ 中，秩的个数是 $F(g) - F(g-1)$ 。显然， $G(N)$ 是最大秩组的一个非常宽松的上界。作为一个例子，假设我们按照图8-17将秩分组。在这种情况下， $G(r) = \lceil \sqrt{r} \rceil$ 。在组 $g$ 中的最大的秩是 $F(g) = g^2$ ，并观察到组 $g > 0$ 且包含秩 $F(g-1) + 1$ 直到 $F(g)$ 。这个公式不适用秩组0，因此为了方便，我们将保证秩组0只包含秩为0的元素。注意，秩组是由一些连续的秩构成的。

| 组   | 秩                      |
|-----|------------------------|
| 0   | 0                      |
| 1   | 1                      |
| 2   | 2, 3, 4                |
| 3   | 5~9                    |
| 4   | 10~16                  |
| $i$ | $((i-1)^2+1) \sim i^2$ |

图8-17 将秩分成秩组的可能的划分

前面提到过，只要每个根记录着它的子树都是多大，则每个union指令仅花费常数时间。因此，就本证明而言，union实际上是不花费代价的。

每个find( $i$ )花费的时间正比于从表示 $i$ 的顶点到根的路径上的顶点的个数。因此，对于路径上的每一个顶点存入一个分币。不过，如果这就是我们所做的全部，那么不能对界有更多的要求，因为没有利用到路径压缩。因此，需要在分析中利用路径压缩。我们将使用想像算账（fancy accounting）的方法。

328

对从表示 $i$ 的顶点到根的路径上的每一个顶点 $v$ ，我们在两个账户之一存入一个分币：

- (1) 如果 $v$ 是根，或者 $v$ 的父亲是根，或者 $v$ 的父亲与 $v$ 在不同的秩组中，那么在该法则之下收取一个单位的费用，这就需要将一个美分币存入公共储金中。
- (2) 否则，将一个加拿大分币存入该顶点中。

**引理8.4** 对于任意的find( $v$ )，不论存入公共储金还是存入顶点，所存分币的总数恰好等于从 $v$ 到根的路径上的结点的个数。

**证明** 这是显而易见的。 ■

如此一来，我们需要做的就是把在法则1下存入的所有美分币和在法则2下存入的所有加拿大分币加起来。

我们最多进行 $M$ 次find。要求出在一次find中能够存入公共储金中的分币个数的界。

**引理8.5** 经过整个算法，在法则1下美分币总的存入量最多为 $M(G(N) + 2)$ 。

**证明** 证明不难。对于任意的find，由于有根和它的儿子，因此存入两个美分币。由引理8.3，沿路径向上分布的结点按秩单调递增，而由于最多有 $G(N)$ 个秩组，因此对任意特定的find，在路径上只有 $G(N)$ 个其他结点能够按照法则1存入分币。于是，在任意一次find期间最多有 $G(N) + 2$ 个美分币可以放入公共储金中。因此，在法则1下，连续 $M$ 次find最多可以存入 $M(G(N) + 2)$ 个美分币。 ■

329 为了得到在法则2下所有加拿大分币存入量的理想的估计值，我们将把按照顶点而不是按照 find 指令所存入的分币量加起来。如果一枚硬币在法则2下存入顶点  $v$ ，那么  $v$  将通过路径压缩被移动并得到具有比它原来的父结点更高的秩的新的父亲（这里我们用到了正在进行路径压缩的事实）。于是，秩组  $g > 0$  中的结点  $v$  在它的父结点被推离秩组  $g$  之前最多可以移动  $F(g) - F(g-1)$  次，因为这是该秩组的大小<sup>1</sup>。在这以后，对  $v$  的所有未来的收费均按照法则1进行。

引理8.6 秩组  $g > 0$  中顶点的个数  $V(g)$  至多为  $N/2^{F(g-1)}$ 。

证明 由引理8.2，至多存在  $N/2^r$  个秩为  $r$  的顶点。对组  $g$  中的秩求和，得到

$$\begin{aligned} V(g) &\leq \sum_{r=F(g-1)+1}^{F(g)} \frac{N}{2^r} \\ &\leq \sum_{r=F(g-1)+1}^{\infty} \frac{N}{2^r} \\ &\leq N \sum_{r=F(g-1)+1}^{\infty} \frac{1}{2^r} \\ &\leq \frac{N}{2^{F(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \\ &\leq \frac{2N}{2^{F(g-1)+1}} \\ &\leq \frac{N}{2^{F(g-1)}} \end{aligned}$$

引理8.7 存入秩组  $g$  的所有顶点的加拿大分币的最大个数至多是  $NF(g)/2^{F(g-1)}$ 。

证明 该秩组的每一个顶点当它的父结点同在该秩组时最多可以接收  $F(g) - F(g-1) \leq F(g)$  个加拿大分币，而引理8.6告诉我们这样的顶点存在的个数。通过简单的乘法可以得到定理的结果。

引理8.8 在法则2下总的存入分币数最多为  $N \sum_{g=1}^{G(N)} F(g)/2^{F(g-1)}$  个加拿大分币。

330 证明 因为秩组0只含有秩为0的元素，所以它不能按照法则2接收分币（这样的元素在该秩组中不可能有父结点）。通过将其他秩组求和则可得到引理指出的界。

这样，我们就得到在法则1和法则2下存入的分币数，该总数为：

$$M(G(N)+2) + N \sum_{g=1}^{G(N)} F(g)/2^{F(g-1)} \quad (8-1)$$

我们还没有指定  $G(N)$  或它的逆  $F(N)$ 。显然，实际上可以自由选择我们想要的任何函数，但是它应使得选择  $G(N)$  极小化上面的界有意义。不过，若是  $G(N)$  太小，则  $F(N)$  就会很大，这将影响到界。一个明显的理想选择是选取  $F(i)$  为由  $F(0) = 0$  和  $F(i) = 2^{F(i-1)}$  递归定义的函数，于是得到  $G(N) = 1 + \lfloor \log^* N \rfloor$ 。图8-18显示了秩是如何由此而划分的。注意，组0只包含秩0，这是前面引理中要求的。 $F$  非常类似于单变量 Ackermann 函数，它们只在基准情形的定义上有所不同（ $F(0) = 1$ ）。

1. 该数可以减1。不过，我们并不刻意简化；此处的界不是经过仔细改进的界。

| 组 | 秩                          |
|---|----------------------------|
| 0 | 0                          |
| 1 | 1                          |
| 2 | 2                          |
| 3 | 3, 4                       |
| 4 | 5~16                       |
| 5 | 17~2 <sup>16</sup>         |
| 6 | 65 537~2 <sup>65 536</sup> |
| 7 | 非常大的秩                      |

图8-18 在证明中用到的将秩分成秩组的实际划分

**定理8.1**  $M$ 次union和find的运行时间为 $O(M \log^* N)$ 。

**证明** 把 $F$ 和 $G$ 的定义插入到式(8-1)中, 美分币的总数为 $O(MG(N)) = O(M \log^* N)$ , 加拿大分币的总数为 $N \sum_{g=1}^{G(N)} F(g)/2^{F(g-1)} = N \sum_{g=1}^{G(N)} 1 = NG(N) = O(N \log^* N)$ 。由于 $M = \Omega(N)$ , 因此得出定理的界。 ■

分析指出, 能够通过路径压缩经常移动的结点很少, 从而总的时间花费相对要少。

## 8.7 一个应用

应用求并/查找数据结构的一个例子是迷宫的生成, 如图8-19所示就是这样一个迷宫。在图8-19中, 起点位于图的左上角, 而终点位于图的右下角。可以把这个迷宫看成是由单元组成的50×80的矩形, 在该矩形中, 左上角的单元被连通到右下角的单元, 而且这些单元与相邻的单元通过墙壁分离开来。

331

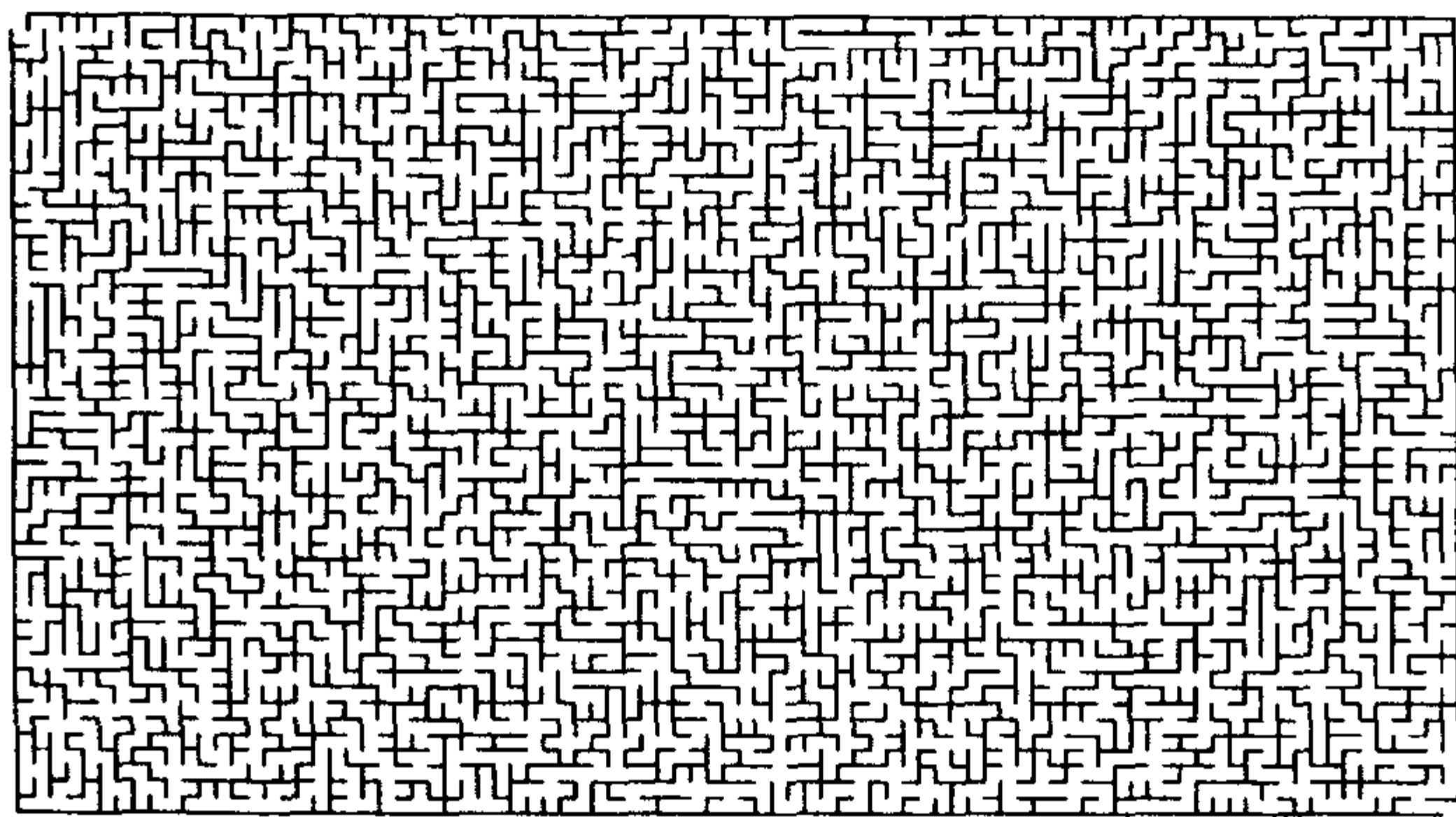


图8-19 一个50×80迷宫

生成迷宫的一个简单算法是从各处的墙壁开始(除入口和出口之外)。此时, 不断地随机选择一面墙, 如果被该墙分割的单元彼此不连通, 那么就把这面墙拆掉。重复这个过程直到开始单元和终止单元连通, 那么就得到一个迷宫。实际上不断地拆掉墙壁直到每一个单元都可以从其他单元达到更好(这会使迷宫产生更多误导的路径)。

我们用5×5迷宫叙述该算法。图8-20显示了初始的状态。我们用union/find数据结构代表彼此互连的单元的集合。开始的时候, 各处都有墙, 而每个单元都在它自己的等价类中。



|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21}  
{22} {23} {24}

图8-20 初始状态：所有的墙都存在，所有的单元都在它自己的集合中

图8-21显示了算法随后的一个状态，这是在一些墙被拆掉之后的状态。设在该阶段连接单元8和13的墙被随机地选做目标。因为单元8和13已经连通（它们在相同的集合中），所以不拆掉这面墙，拆掉它使得迷宫简单化了。设单元18和13是随机选出的下一个目标。通过执行两次find操作可以看到它们是在不同的集合中；因此单元18和13还没有连通。于是把隔开它们的墙拆掉，如图8-22所示。注意，这次操作的结果是包含18和13的两个集合通过union操作被连在一起。这是因为连通到单元18的每个单元现在已与连通到13的每个单元连通。该算法结束时每个单元之间都是连通的，如图8-23所示，构建迷宫的工作完成。

332

|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

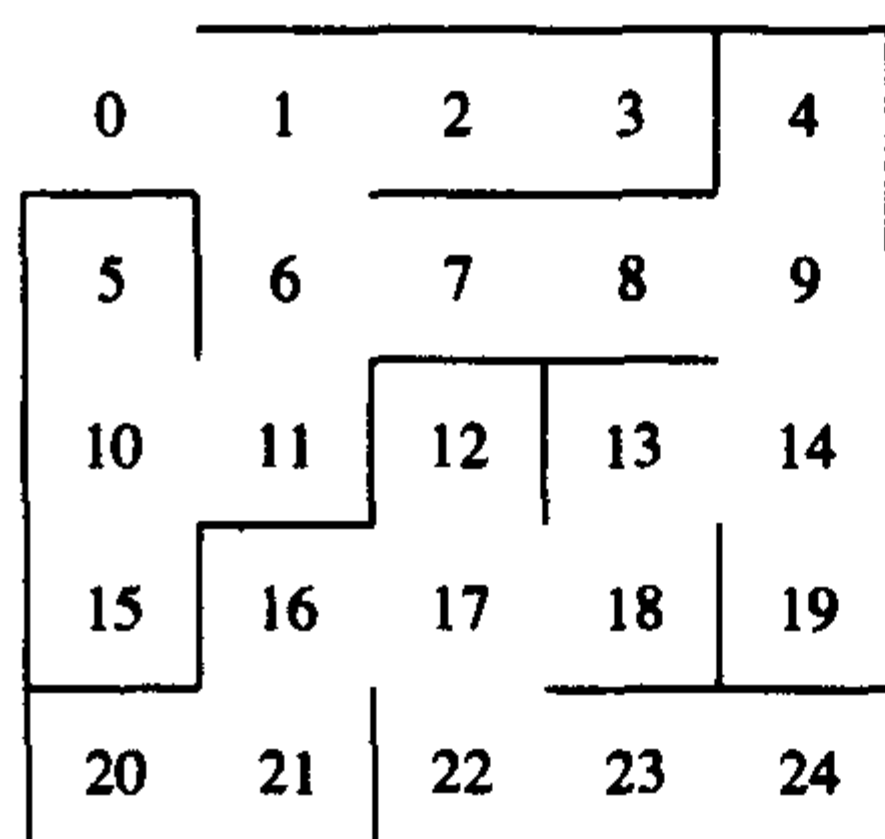
{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14} {5} {10, 11, 15} {12} {16, 17, 18, 22} {19} {20} {21} {23} {24}

图8-21 在算法的某个时刻：几面墙被拆掉，集合合并。如果这时单元8和13之间的墙被随机地选定，那么这面墙将不拆掉，因为单元8和13已经是连通的

|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {5} {10, 11, 15} {12} {19} {20} {21} {23} {24}

图8-22 在图8-21中单元18和13之间的墙被随机地选定。这面墙被拆掉，因为单元18和13还没有连通。它们所在的集合被合并



{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}

图8-23 最后, 24面墙被拆掉, 所有的元素都在一个集合中

这个算法的运行时间由union/find的开销控制。union/find总体的大小等于单元的个数。find操作的次数与单元的个数成正比, 因为拆掉的墙的数目比单元的个数少1, 而仔细观察可以发现, 开始的时候墙的数目大约只有单元个数的两倍。因此, 如果 $N$ 是单元的个数, 由于每面随机选择的墙有两次find, 那么整个算法中find操作的次数估计在 $2N$ 和 $4N$ 之间。因此算法的运行时间可以取为 $O(M \log^* N)$ , 利用这个算法将会很快地生成一个迷宫。

333

## 小结

我们已经看到保持不相交集的非常简单的数据结构。当执行union操作时, 就正确性而言, 哪个集合保留它的名字是无关紧要的。这里, 有需要注意, 当某一特定的步骤尚未完全指定的时候, 考虑选择方案可能是非常重要的。union是灵活的; 利用这一点, 我们能够得到一个有效得多的算法。

路径压缩是自调整 (self-adjustment) 的最早形式之一, 我们已经在其他一些地方 (伸展树、斜堆) 见到过。它的使用非常有趣, 特别是从理论的观点来看, 因为它是算法简单但最坏情形分析却并不简单的第一批例子之一。

334

## 练习

- 8.1 指出下列一系列指令的结果: union(1,2), union(3,4), union(3,5), union(1,7), union(3,6), union(8,9), union(1,8), union(3,10), union(3,11), union(3,12), union(3,13), union(14,15), union(16,0), union(14,16), union(1,3), Union(1,14), 当union是
  - a. 任意进行的。
  - b. 按高度进行的。
  - c. 按大小进行的。
- 8.2 对于上题中的每一棵树, 用对最深结点的路径压缩执行一次find。
- 8.3 编写一个程序来确定路径压缩法和各种求并方法的效果。你的程序应该使用所有六种可能的方法处理一系列等价操作。
- 8.4 证明: 如果union按照高度进行, 那么任意一棵树的深度为 $O(\log N)$ 。
- 8.5
  - a. 证明: 如果 $M = N^2$ , 那么 $M$ 次union/find操作的运行时间是 $O(M)$ 。
  - b. 证明: 如果 $M = N \log N$ , 那么 $M$ 次union/find操作的运行时间是 $O(M)$ 。

- \*c. 设  $M = \Theta(N \log \log N)$ , 则  $M$  次 union/find 操作的运行时间是多少?
- \*d. 设  $M = \Theta(N \log^* N)$ , 则  $M$  次 union/find 操作的运行时间是多少?
- 8.6 证明: 对于由 8.7 节中的算法生成的迷宫, 从开始点到终止点的路径是唯一的。
- 8.7 设计一个生成迷宫的算法, 这个迷宫不含有从开始点到终止点的路径, 但却有一个性质, 即拆除预先指定的一面墙后则建立一条唯一的路径。
- \*8.8 假设我们想要添加一个附加的操作 deunion, 它废除尚未被废除的最后的 union 操作。
- 证明: 如果我们按高度求并以及不用路径压缩进行 find, 那么 deunion 操作容易进行并且连续  $M$  次 union、find 和 deunion 操作花费  $O(M \log N)$  时间。
  - 为什么路径压缩使得 deunion 很难进行?
- \*\*c. 指出如何实现所有三种操作使得连续  $M$  次操作花费  $O(M \log N / \log \log N)$  时间。
- \*8.9 假设我们想要添加一种额外的操作 remove( $x$ ), 该操作把  $x$  从当前的集合中除去并把它放到它自己的集合中。指出如何修改 union/find 算法使得连续  $M$  次 union、find 和 remove 操作的运行时间为  $O(M \alpha(M, N))$ 。
- \*\*8.10 给出一个算法, 以一棵  $N$  顶点树和  $N$  对顶点作为输入, 对每对顶点 ( $v, w$ ) 确定  $v$  和  $w$  的最近的公共祖先。你的算法应该以  $O(N \log^* N)$  时间运行。
- \*8.11 证明: 如果所有的 union 都在 find 之前, 那么使用路径压缩的不相交集算法需要线性时间, 即使 union 是任意进行的也是如此。
- \*\*8.12 证明: 如果 union 操作任意进行, 但路径压缩是对 find 进行, 那么最坏情形运行时间为  $\Theta(M \log N)$ 。
- 8.13 证明: 如果 union 按大小进行且执行路径压缩, 那么最坏情形运行时间为  $O(M \log^* N)$ 。
- 8.14 设我们通过使得从  $i$  到根的路径上的每一个其他结点都链接到它的祖父 (当有意义时) 以实现 find( $i$ ) 的偏路径压缩 (partial path compression)。这叫作路径平分 (path halving)。
- 编写一个过程完成上述工作。
  - 证明: 如果对 find 操作进行路径平分, 则不论使用按高度求并还是按大小求并, 其最坏情形运行时间皆为  $O(M \log^* N)$ 。
- 8.15 编写一个能够生成任意大小的迷宫的程序。如果你所使用的系统支持图形界面, 那么, 生成一个类似于图 8-19 那样的迷宫。否则, 给出一个文本形式的迷宫 (例如, 输出的每条线都代表一个矩形, 并且给出相应的哪面墙存在的信息) 并用程序实现。

## 参考文献

求解求并/查找问题的各种方案可以在 [6]、[9] 和 [11] 中找到。Hopcroft 和 Ullman 证明了 8.6 节的  $O(M \log^* N)$  界。Tarjan [15] 则得到界  $O(M \alpha(M, N))$ 。对于  $M < N$  的更精确 (但渐近恒等) 的界见于 [2] 和 [18]。对路径压缩和 union 的各种其他方法也达到相同的界; 详细细节见 [18]。

由 Tarjan [16] 给出的一个下界指出, 在一定的限制下处理  $M$  次 union/find 操作需要  $\Omega(M \alpha(M, N))$  时间。在一些较少的限制条件下 [7] 和 [14] 指出相同的界。

求并/查找数据结构的应用见于 [1] 和 [10]。求并/查找问题的某些特殊情形可以以  $O(M)$  时间解决, 见 [8]。这使得若干算法的运行时间得以降低一个  $\alpha(M, N)$  因子, 如 [1]、图优势 (graph dominance) 以及可约性 (见第 9 章的参考文献)。另外一些像 [10] 和本章中的图连通性问题等并未受到影响。文章列举了 10 个例子。Tarjan 还对若干图论问题使用路径压缩得到一些有效的算法 [17]。

求并/查找问题平均情形的一些结果见于 [5]、[12]、[21] 和 [3]。不同于整个运算序列, 一些为任意单次操作确定运行时间的界的结果可在 [4] 和 [13] 中找到。

练习 8.8 在 [20] 中解决。一般的求并/查找结构在 [19] 中给出, 这种结构支持更多的操作。

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "On Finding Lowest Common Ancestors in Trees," *SIAM Journal on Computing*, 5 (1976), 115-132.
2. L. Banachowski, "A Complement to Tarjan's Result about the Lower Bound on the Complexity of the Set Union Problem," *Information Processing Letters*, 11 (1980), 59-65.
3. B. Bollobás and I. Simon, "Probabilistic Analysis of Disjoint Set Union Algorithms," *SIAM Journal on Computing*, 22 (1993), 1053-1086.
4. N. Blum, "On the Single-Operation Worst-Case Time Complexity of the Disjoint Set Union Problem," *SIAM Journal on Computing*, 15 (1986), 1021-1024.
5. J. Doyle and R. L. Rivest, "Linear Expected Time of a Simple Union Find Algorithm," *Information Processing Letters*, 5 (1976), 146-148.
6. M.J. Fischer, "Efficiency of Equivalence Algorithms," in *Complexity of Computer Computation* (eds. R. E. Miller and J. W. Thatcher), Plenum Press, New York, 1972, 153-168.
7. M. L. Fredman and M. E. Saks, "The Cell Probe Complexity of Dynamic Data Structures," *Proceedings of the Twenty-first Annual Symposium on Theory of Computing* (1989), 345-354.
8. H. N. Gabow and R. E. Tarjan, "A Linear-Time Algorithm for a Special Case of Disjoint Set Union," *Journal of Computer and System Sciences*, 30 (1985), 209-221.
9. B. A. Galler and M. J. Fischer, "An Improved Equivalence Algorithm," *Communications of the ACM*, 7 (1964), 301-303.
10. J. E. Hopcroft and R. M. Karp, "An Algorithm for Testing the Equivalence of Finite Automata," *Technical Report TR-71-114*, Department of Computer Science, Cornell University, Ithaca, N.Y., 1971.
11. J. E. Hopcroft and J. D. Ullman, "Set Merging Algorithms," *SIAM Journal on Computing*, 2 (1973), 294-303.
12. D. E. Knuth and A. Schonhage, "The Expected Linearity of a Simple Equivalence Algorithm," *Theoretical Computer Science*, 6 (1978), 281-315.
13. J. A. LaPoutre, "New Techniques for the Union-Find Problem," *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), 54-63.
14. J. A. LaPoutre, "Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines," *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing* (1990), 34-44.
15. R. E. Tarjan, "Efficiency of a Good but Not Linear Set Union Algorithm," *Journal of the ACM*, 22 (1975), 215-225.
16. R. E. Tarjan, "A Class of Algorithms Which Require Nonlinear Time to Maintain Disjoint Sets," *Journal of Computer and System Sciences*, 18 (1979), 110-127.
17. R. E. Tarjan, "Applications of Path Compression on Balanced Trees," *Journal of the ACM*, 26 (1979), 690-715.
18. R. E. Tarjan and J. van Leeuwen, "Worst Case Analysis of Set Union Algorithms," *Journal of the ACM*, 31 (1984), 245-281.
19. M. J. van Kreveld and M. H. Overmars, "Union-Copy Structures and Dynamic Segment Trees," *Journal of the ACM*, 40 (1993), 635-652.
20. J. Westbrook and R. E. Tarjan, "Amortized Analysis of Algorithms for Set Union with Backtracking," *SIAM Journal on Computing*, 18 (1989), 1-11.
21. A. C. Yao, "On the Average Behavior of Set Merging Algorithms," *Proceedings of Eighth Annual ACM Symposium on the Theory of Computation* (1976), 192-195.

336

337



**本章**讨论图论中的几个常见问题。这些算法不仅在实践中有用，而且还是非常有趣的，因为在许多实际应用中若不仔细选择数据结构将导致速度过慢。本章将：

- 介绍几个现实生活中发生的问题，它们可以转化成图论问题。
- 给出一些算法以解决几个常见的图论问题。
- 指出适当选择数据结构可以极大地降低这些算法的运行时间。
- 介绍一个被称为深度优先搜索（depth-first search）的重要技巧，并指出它如何能够以线性时间求解若干表面上非平凡的问题。

## 9.1 若干定义

图（graph） $G = (V, E)$ 由顶点（vertex）的集 $V$ 和边（edge）的集 $E$ 组成。每一条边就是一个点对 $(v, w)$ ，其中 $v, w \in V$ 。有时也把边称作弧（arc）。如果点对是有序的，那么图就叫作有向的图。有向的图有时也叫作有向图（digraph）。顶点 $v$ 和 $w$ 邻接（adjacent）当且仅当 $(v, w) \in E$ 。在一个具有边 $(v, w)$ 从而具有边 $(w, v)$ 的无向图中， $w$ 和 $v$ 邻接且 $v$ 也和 $w$ 邻接。有时候边还具有第三种成分，称作权（weight）或值（cost）。

图中的一条路径（path）是一个顶点序列 $w_1, w_2, w_3, \dots, w_N$ ，其中 $(w_i, w_{i+1}) \in E$ ， $1 \leq i < N$ 。这样一条路径的长（length）是该路径上的边数，它等于 $N-1$ 。从一个顶点到它自身可以看成是一条路径；如果路径不包含边，那么路径的长为0。这是定义特殊情形的一种方便的方法。如果图含有一条从一个顶点到它自身的边 $(v, v)$ ，那么路径 $v, v$ 有时候也叫作环（loop）。我们要讨论的图一般是无环的。简单路径是指路径上的所有顶点都是互异的，但第一个顶点和最后一个顶点可能相同。

有向图中的回路（cycle）是满足 $w_1 = w_N$ 且长至少为1的一条路径；如果该路径是简单路径，那么这个回路就是简单回路。对于无向图，我们要求边是互异的。这些要求的根据在于无向图中的路径 $u, v, u$ 不应该被认为是回路，因为 $(u, v)$ 和 $(v, u)$ 是同一条边。但是在有向图中它们是两条不同的边，因此称它们为回路是有意义的。如果一个有向图没有回路，则称其为无环的（acyclic）。有向无环图有时也简称为DAG。

**339** 如果在无向图中从每一个顶点到每个其他顶点都存在一条路径，则称该无向图是连通的（connected）。具有这样性质的有向图称为是强连通的（strongly connected）。如果有向图不是强连通的，但是它的基础图（underlying graph，即其弧上去掉方向所形成的图）是连通的，那么称该有向图为弱连通的（weakly connected）。完全图（complete graph）是其每一对顶点间都存在一条边的图。

现实生活中能够用图进行模拟的一个例子是航空系统。每个机场是一个顶点，在由两个顶点

表示的机场间如果存在一条直达航线，那么这两个顶点就用一条边连接。边可以有一个权，表示时间、距离或飞行的费用。有理由假设，这样的图是有向图，因为在不同的方向上飞行可能所用时间或所花的费用会不同（例如，依赖于地方税）。可能我们更希望航空系统是强连接的，这样就总能够从任一机场飞到另外的任意一个机场。我们也可能愿意迅速确定任意两个机场之间的最佳航线。“最佳”可以是指最少边数的路径，也可以是对一种或所有权重重量度所算出的最佳者。

交通流也可以用图来模型化。每一条街道交叉口表示一个顶点，而每一条街道就是一条边。边的值可能代表速度限度或是容量（车道的数目）等等。此时我们可能需要找出一条最短路线，或用该信息找出最可能产生交通瓶颈的位置。

本章的其余部分将考察图论的几个其他应用，其中有许多可能是相当巨大的，因此，所使用算法的效率是非常重要的。

## 图的表示

我们将考虑有向图（无向图可类似表示）。

现在假设可以从1开始对顶点编号。图9-1表示了7个顶点和12条边。

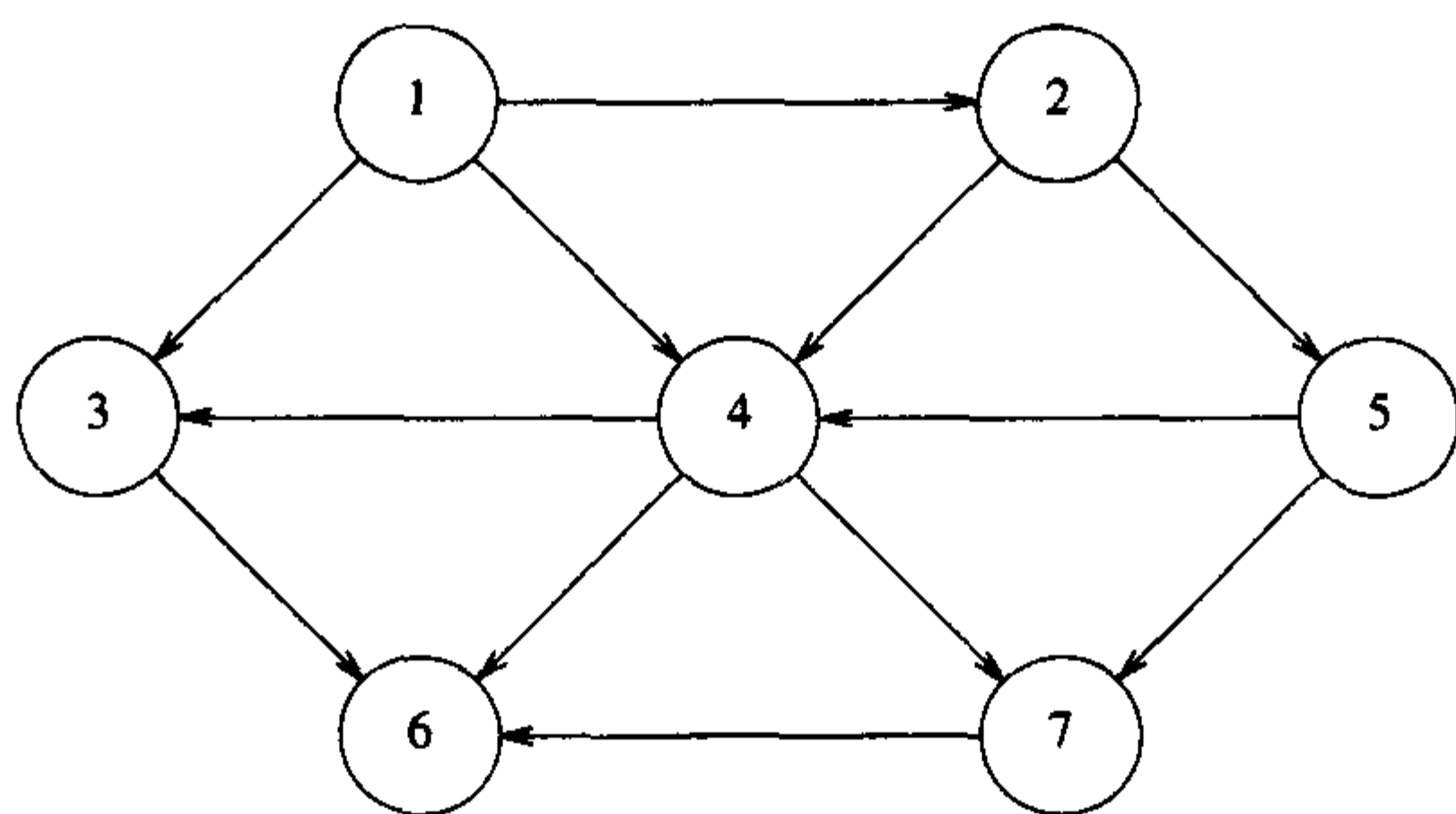


图9-1 一个有向图

表示图的一种简单方法是使用二维数组，称为邻接矩阵（adjacent matrix）表示法。对于每条边 $(u, v)$ ，置 $A[u][v] = \text{true}$ ；否则，数组的项就是false。如果边有一个权，那么可以置 $A[u][v]$ 等于该权，而使用一个很大或者很小的权作为标记表示不存在的边。例如，如果要寻找最便宜的航空路线，那么可以用值 $\infty$ 来表示不存在的航线。如果出于某种原因要寻找最昂贵的航线，那么可以用 $-\infty$ （或者使用0）来表示不存在的边。

340

虽然这种表示的优点是非常简单，但是，它的空间需求则为 $\Theta(|V|^2)$ ，如果图的边不是很多，那么这种表示的代价就太大了。若图是稠密的（dense）： $|E| = \Theta(|V|^2)$ ，则邻接矩阵是合适的表示方法。不过，在我们将要看到的大部分应用中，情况并不如此。例如，设用图表示一个街道地图，街道呈曼哈顿式，其中几乎所有的街道或者南北向、或者东西向。因此，任一路口大致都有四条街道，于是，如果图是有向图且所有的街道都是双向的，则 $|E| \approx 4|V|$ 。如果有3000个路口，那么就得到一个3000顶点的图，该图有12 000条边，它们需要一个大小为9 000 000的数组。该数组的大部分项将是0。这从直观来看很糟，因为我们想要数据结构表示那些实际存在的数据，而不是去表示不存在的数据。

如果图不是稠密的，换句话说，是稀疏的（sparse），则更好的解决方法是使用邻接表（adjacency list）表示。对每一个顶点，我们使用一个表存放所有邻接的顶点。此时的空间需求为 $O(|E| + |V|)$ ，

它相对于图的大小而言是线性的<sup>1</sup>。图9-2最左边的结构只是头单元（header cell）的数组。这种表示方法从图9-2可以清楚地看出。如果边有权，那么这个附加的信息也可以存储在单元中。

|   |         |
|---|---------|
| 1 | 2, 4, 3 |
| 2 | 4, 5    |
| 3 | 6       |
| 4 | 6, 7, 3 |
| 5 | 4, 7    |
| 6 | (空)     |
| 7 | 6       |

图9-2 图的邻接表表示

邻接表是表示图的标准方法。无向图可以类似地表示；每条边 $(u, v)$ 出现在两个表中，因此空间的使用基本上是双倍的。在图论算法中通常需要找出与某个给定顶点 $v$ 邻接的所有的顶点。而这可以通过简单地扫描相应的邻接表来完成，所用时间与这些找到的顶点的个数成正比。

有很多方法可以维护邻接表。首先，要注意到这些表自身可以通过vector或list来维护。然而，对于稀疏图，当使用vector时，程序员需要将每一个vector都初始化为比默认值稍小一点的容量，否则就会浪费大量的空间。

由于快速得到任何顶点的邻接顶点列表很重要，所以有两个基本的选择：或者使用图（map），其键为顶点，值为邻接表，或者将每个邻接表作为Vertex类的成员函数处理。第一个选择似乎简单些，但是第二个选择更快，因为避免了在图中的重复查找。

对第二种情况，如果顶点是string（例如机场的名字或者路口的名字），那么可以使用图。其中，键是顶点的名字，值是Vertex（典型的是一个指向Vertex的指针），并且每一个Vertex对象都有一个邻接顶点（指向顶点的指针）的列表，可能还包括原始的string名。

在本章中将尽可能使用伪代码来描述图论算法。这么做将节省空间，当然，也使得算法的运算表达式更清晰。在9.3节的最后，提供了一个可用的C++实现的例程。该例程基于最短路径算法获得答案。

341

## 9.2 拓扑排序

拓扑排序（topological sort）是对有向无环图的顶点的一种排序，它使得如果存在一条从 $v_i$ 到 $v_j$ 的路径，那么在排序中 $v_j$ 出现在 $v_i$ 的后面。图9-3表示了迈阿密州立大学的课程结构。有向边 $(v, w)$ 表明课程 $v$ 必须在课程 $w$ 选修前修完。这些课程的拓扑排序不会破坏课程结构要求的任意课程序列。

1. 在我们谈到线性时间图论算法的时候，要求运行时间为  $O(|E| + |V|)$ 。

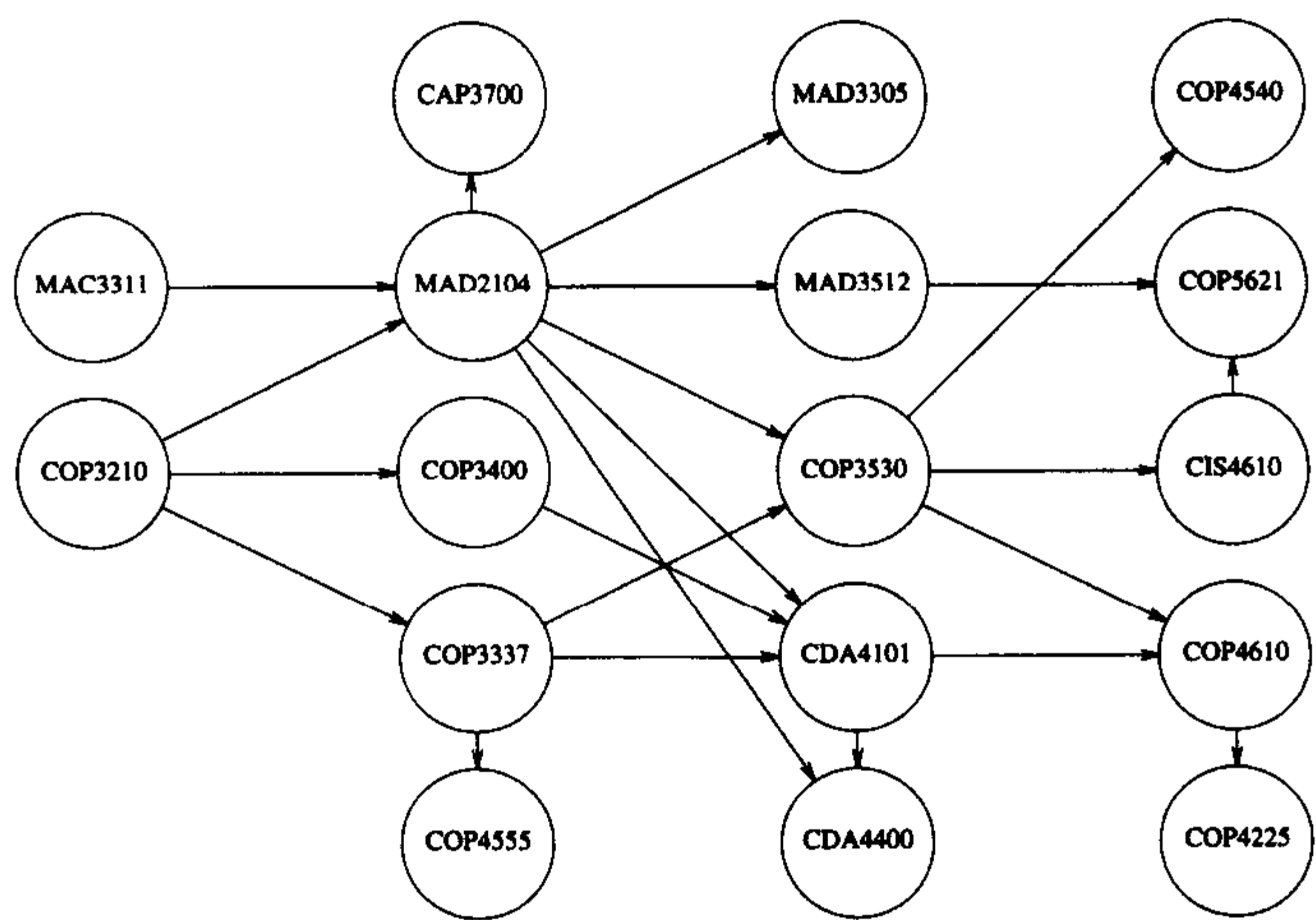


图9-3 表示课程结构的无环图

显然，如果图含有回路，那么拓扑排序是不可能的，因为对于回路上的两个顶点 $v$ 和 $w$ ， $v$ 先于 $w$ 同时 $w$ 又先于 $v$ 。此外，排序不必是唯一的；任何合理的排序都是可以的。在图9-4中， $v_1, v_2, v_3, v_4, v_3, v_7, v_6$ 和 $v_1, v_2, v_5, v_4, v_7, v_3, v_6$ 都是拓扑排序。

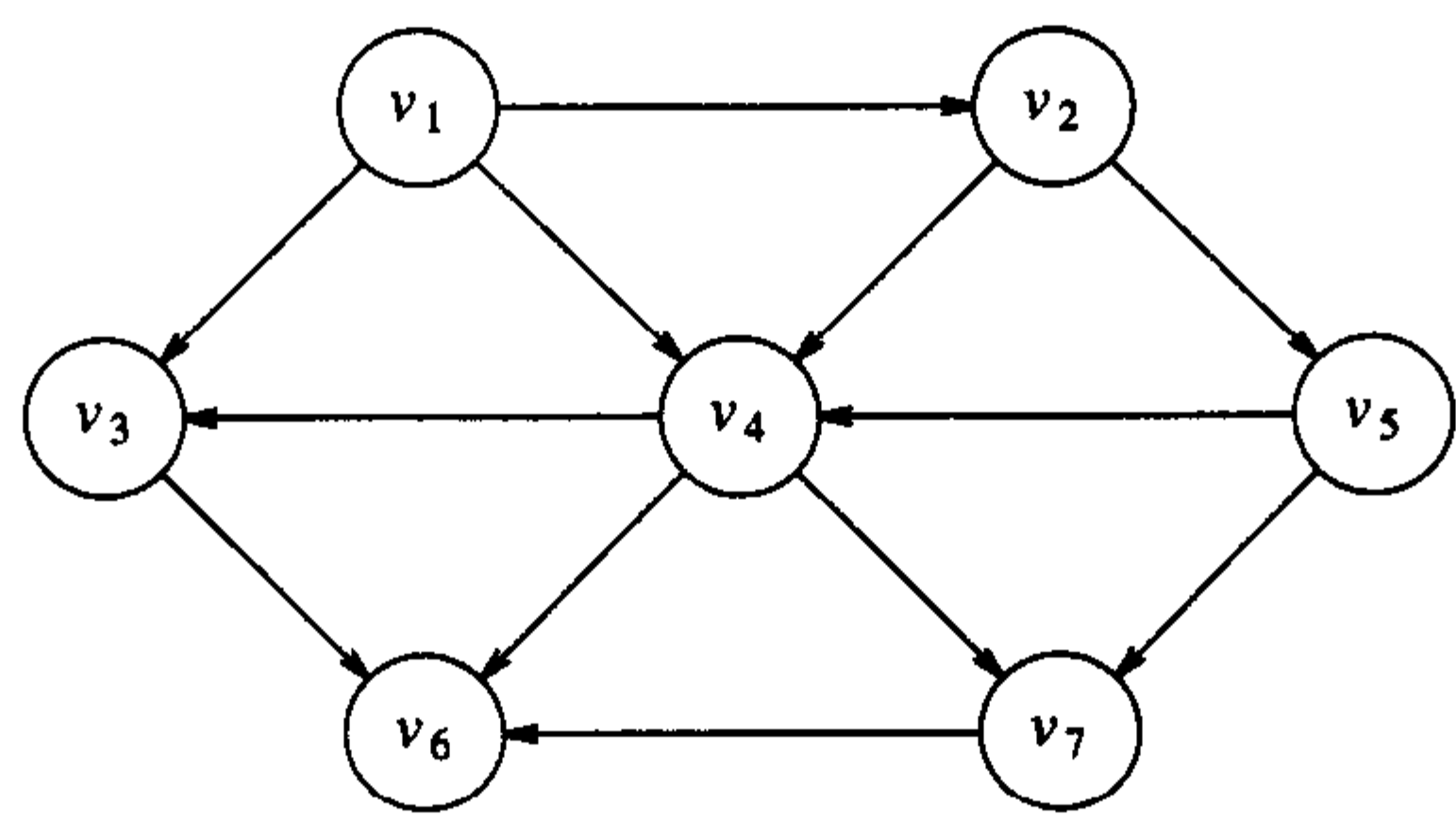


图9-4 一个无环图

一个简单的求拓扑排序的算法是先找出任意一个没有入边的顶点。然后显示出该顶点，并将它和它的边一起从图中删除。然后，对图的其余部分应用同样的方法处理。

为了将上述方法公式化，我们把顶点 $v$ 的入度（indegree）定义为边 $(u, v)$ 的条数。计算图中所有顶点的入度。假设每一个顶点的入度被存储且图被读入一个邻接表中，则此时可以应用图9-5中的算法生成一个拓扑排序。

```
void Graph::topsort( )
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == NOT_A_VERTEX )
            throw CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

图9-5 简单的拓扑排序伪代码



函数findNewVertexOfIndegreeZero扫描数组，寻找一个尚未被分配拓扑编号的入度为0的顶点。如果不存在这样的顶点，那么它返回NOT\_A\_VERTEX；这就意味着该图有回路。

因为findNewVertexOfIndegreeZero函数是对顶点的数组的一个简单的顺序扫描，所以每次对它的调用都花费 $O(|V|)$ 时间。由于有 $|V|$ 次这样的调用，因此该算法的运行时间为 $O(|V|^2)$ 。

通过更仔细地选择数据结构可以做得更好。产生如此差的运行时间的原因在于对顶点的数组的顺序扫描。如果图是稀疏的，那么在每次迭代期间只有少数顶点的入度被更新。然而，虽然只有一小部分发生变化，但在搜索入度为0的顶点时我们（潜在地）查看了所有的顶点。

可以通过将所有（未分配拓扑编号）的入度为0的顶点放在一个特殊的盒子中而避免这种无效的劳动。此时findNewVertexOfIndegreeZero函数返回（并删除）盒子中的任一顶点。当降低邻接顶点的入度时，检查每一个顶点并在入度降为0时把它放入盒子中。

为实现这个盒子，可以使用一个栈或队列。这里使用队列。首先，计算每个顶点的入度。然后，将所有入度为0的顶点放入一个初始为空的队列中。当队列不空时，删除一个顶点 $v$ ，并将与 $v$ 邻接的所有顶点的入度减1。只要顶点的入度降为0，就把该顶点放入队列中。此时，拓扑排序就是顶点出队的顺序。图9-6显示了每一阶段之后的状态。

| 顶点    | 出队前的入度 |       |       |       |            |       |       |
|-------|--------|-------|-------|-------|------------|-------|-------|
|       | 1      | 2     | 3     | 4     | 5          | 6     | 7     |
| $v_1$ | 0      | 0     | 0     | 0     | 0          | 0     | 0     |
| $v_2$ | 1      | 0     | 0     | 0     | 0          | 0     | 0     |
| $v_3$ | 2      | 1     | 1     | 1     | 0          | 0     | 0     |
| $v_4$ | 3      | 2     | 1     | 0     | 0          | 0     | 0     |
| $v_5$ | 1      | 1     | 0     | 0     | 0          | 0     | 0     |
| $v_6$ | 3      | 3     | 3     | 3     | 2          | 1     | 0     |
| $v_7$ | 2      | 2     | 2     | 1     | 0          | 0     | 0     |
| 入队    | $v_1$  | $v_2$ | $v_5$ | $v_4$ | $v_3, v_7$ |       | $v_6$ |
| 出队    | $v_1$  | $v_2$ | $v_5$ | $v_4$ | $v_3$      | $v_7$ | $v_6$ |

图9-6 对图9-4中的图应用拓扑排序的结果

这个算法的伪代码实现在图9-7中给出。和前面一样，我们将假设图已经被读到邻接表中，且入度已计算并和顶点一起存储。我们还假设每个顶点有一个字段，叫作topNum，其中存放的是拓扑编号。

如果使用邻接表，那么执行这个算法所用的时间为 $O(|E|+|V|)$ 。当认识到for循环体对每条边最多执行一次时，这个结果是显而易见的。队列操作对每个顶点最多进行一次，而初始化各步花费的时间也和图的大小成正比。

### 9.3 最短路径算法

本节我们考察各种最短路径问题。输入是一个加权图：与每条边 $(v_i, v_j)$ 相联系的是穿越该边的代价（或称为值） $c_{i,j}$ 。一条路径 $v_1v_2\dots v_N$ 的值是 $\sum_{i=1}^{N-1} c_{i,i+1}$ ，叫作加权路径长（weighted path length）。

而无权路径长（unweighted path length）只是路径上的边数，即 $N-1$ 。

```

void Graph::topsort( )
{
    Queue<Vertex> q;
    int counter = 0;

    q.makeEmpty( );
    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter; // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw CycleFoundException( );
}

```

图9-7 施行拓扑排序的伪代码

**单源最短路径问题** 给定一个加权图 $G = (V, E)$ 和一个特定顶点 $s$ 作为输入，找出从 $s$ 到 $G$ 中每一个其他顶点的最短加权路径。

例如，在图9-8中，从 $v_1$ 到 $v_6$ 的最短加权路径的值为6，它是从 $v_1$ 到 $v_4$ 到 $v_7$ 再到 $v_6$ 的路径。在这两个顶点间的最短无权路径长为2。一般说来，当不指明所讨论的是加权路径还是无权路径时，如果图是加权的，那么路径就是加权的。还要注意，在图9-8中，从 $v_6$ 到 $v_1$ 没有路径。

前面例子中的图没有负值的边。图9-9指出负边的问题可能产生。从 $v_5$ 到 $v_4$ 的路径的值为1，但是，通过下面的循环 $v_5, v_4, v_2, v_5, v_4$ 存在一条更短的路径，它的值是-5。这条路径仍然不是最短的，因为可以在循环中滞留任意长。因此，这两个顶点间的最短路径问题是不确定的。类似地，从 $v_1$ 到 $v_6$ 的最短路径也是不确定的，因为可以进入同样的循环。这个循环叫作**负值回路** (negative-cost cycle)；当它出现在图中时，最短路径问题就是不确定的。有负值的边未必不好，但是它们的出现似乎使问题增加了难度。为方便起见，在没有负值回路时，从 $s$ 到 $s$ 的最短路径为0。

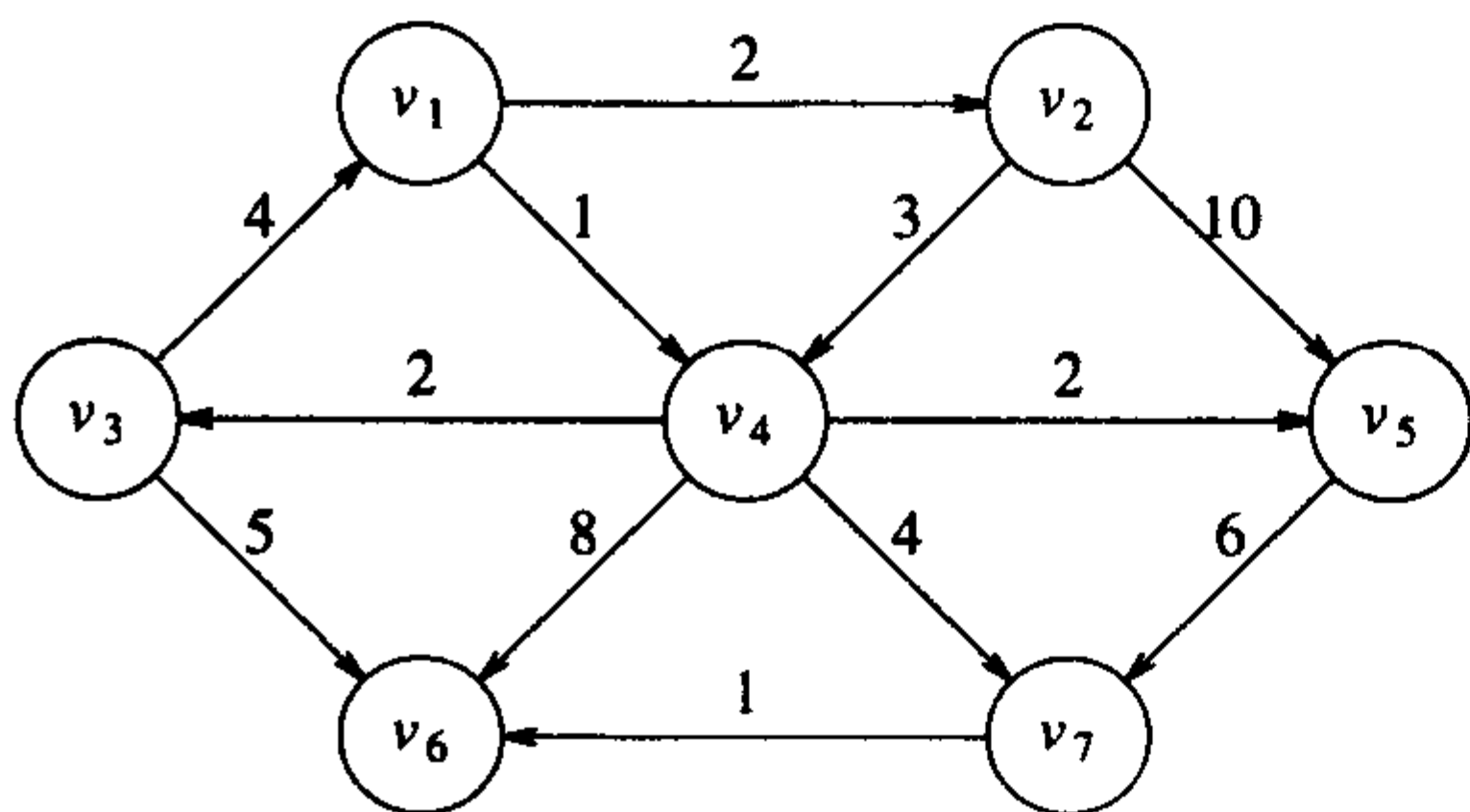


图9-8 有向图G

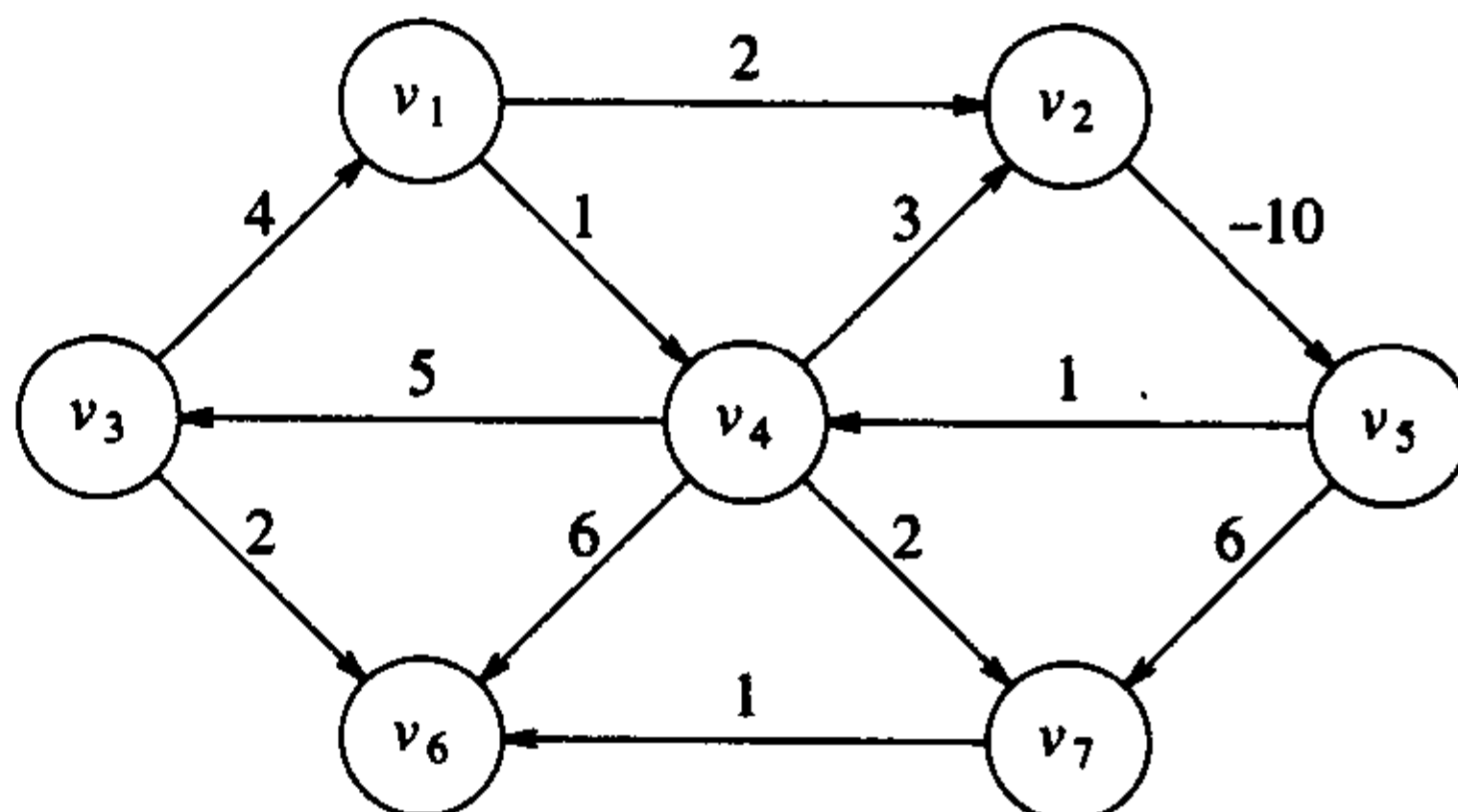


图9-9 带有负值回路的图

有许多例子可能要求解最短路径问题。如果顶点代表计算机，边代表计算机间的链接，值表示通信的花费（每1000字节数据的电话费）、延迟成本（传输1000字节所需要的秒数）或它们及其他因素的组合，那么可以利用最短路径算法来找出从一台计算机向一组其他计算机发送电子

新闻的最便宜的方法。

可以使用图为航线或其他大规模运输路线建立模型并利用最短路径算法计算两点间的最佳路线。在类似这样的许多实际应用中，我们可能想要找出从一个顶点 $s$ 到另一个顶点 $t$ 的最短路径。当前，还不存在找出从 $s$ 到一个顶点的路径比找出从 $s$ 到所有顶点的路径更快（快得超出一个常数因子）的算法。

我们将考察求解该问题四种形态的算法。首先，考虑无权最短路径问题并指出如何以 $O(|E|+|V|)$ 时间解决它。其次，我们还要介绍，假设没有负边，那么如何求解加权最短路径问题。这个算法在使用合理的数据结构实现时，运行时间为 $O(|E|\log|V|)$ 。

如果图有负边，我们将提供一个简单的解法，不过它的时间界不理想，为 $O(|E|\cdot|V|)$ 。最后，我们将以线性时间解决无环图这种特殊情形的加权问题。

### 9.3.1 无权最短路径

图9-10表示一个无权图 $G$ 。使用某个顶点 $s$ 作为输入参数，我们想要找出从 $s$ 到所有其他顶点的最短路径。我们只对包含在路径中的边数感兴趣，因此在边上不存在权。显然，这是加权最短路径问题的特殊情形，因为可以为所有的边都赋以权1。

暂时假设我们只对最短路径的长而不是具体的路径本身感兴趣。记录实际的路径只不过是简单的簿记问题。

设选择 $s$ 为 $v_3$ 。此时立刻可以看出从 $s$ 到 $v_3$ 的最短路径是长为0的路径。把这个信息做个标记，得到图9-11。

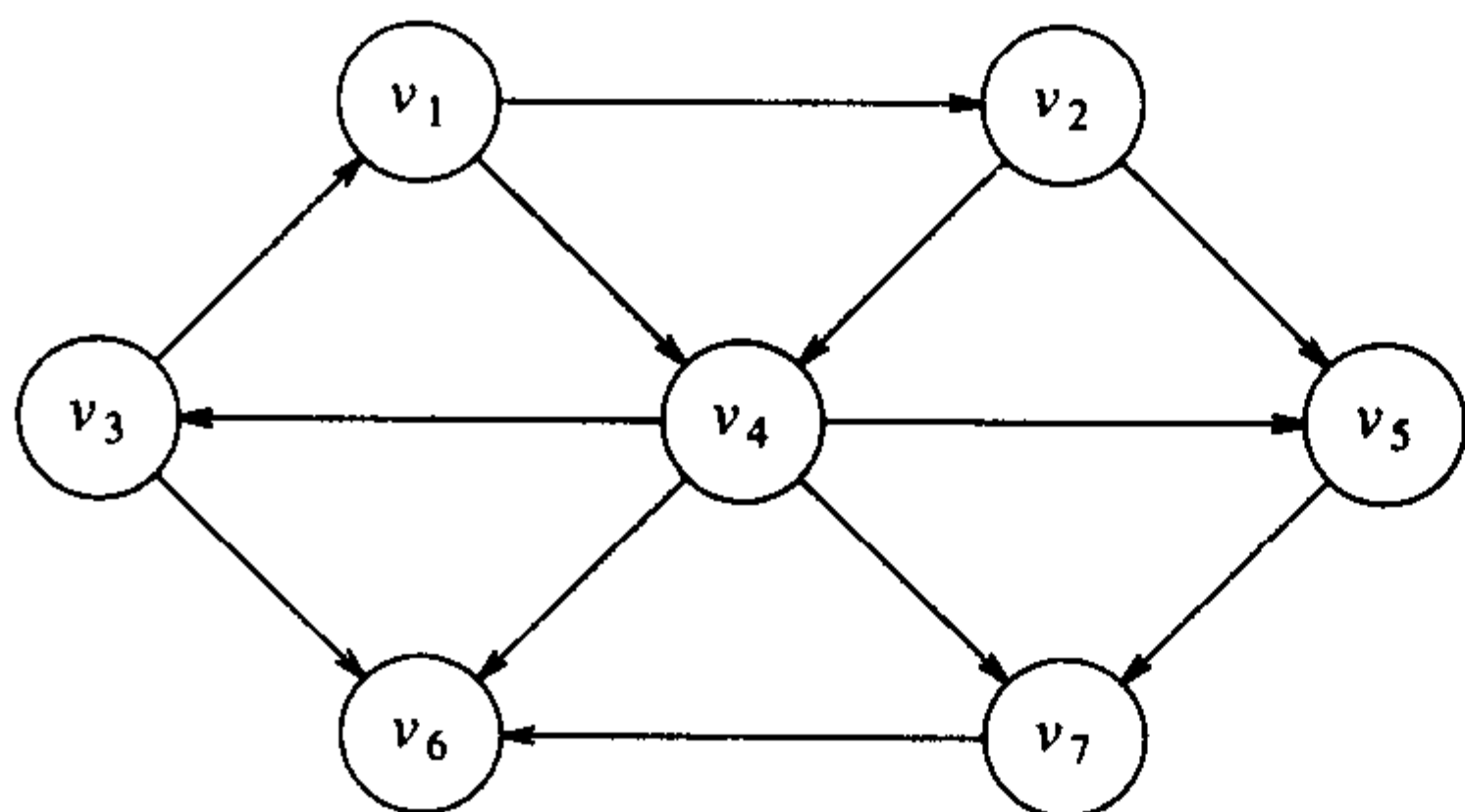


图9-10 一个无权有向图 $G$

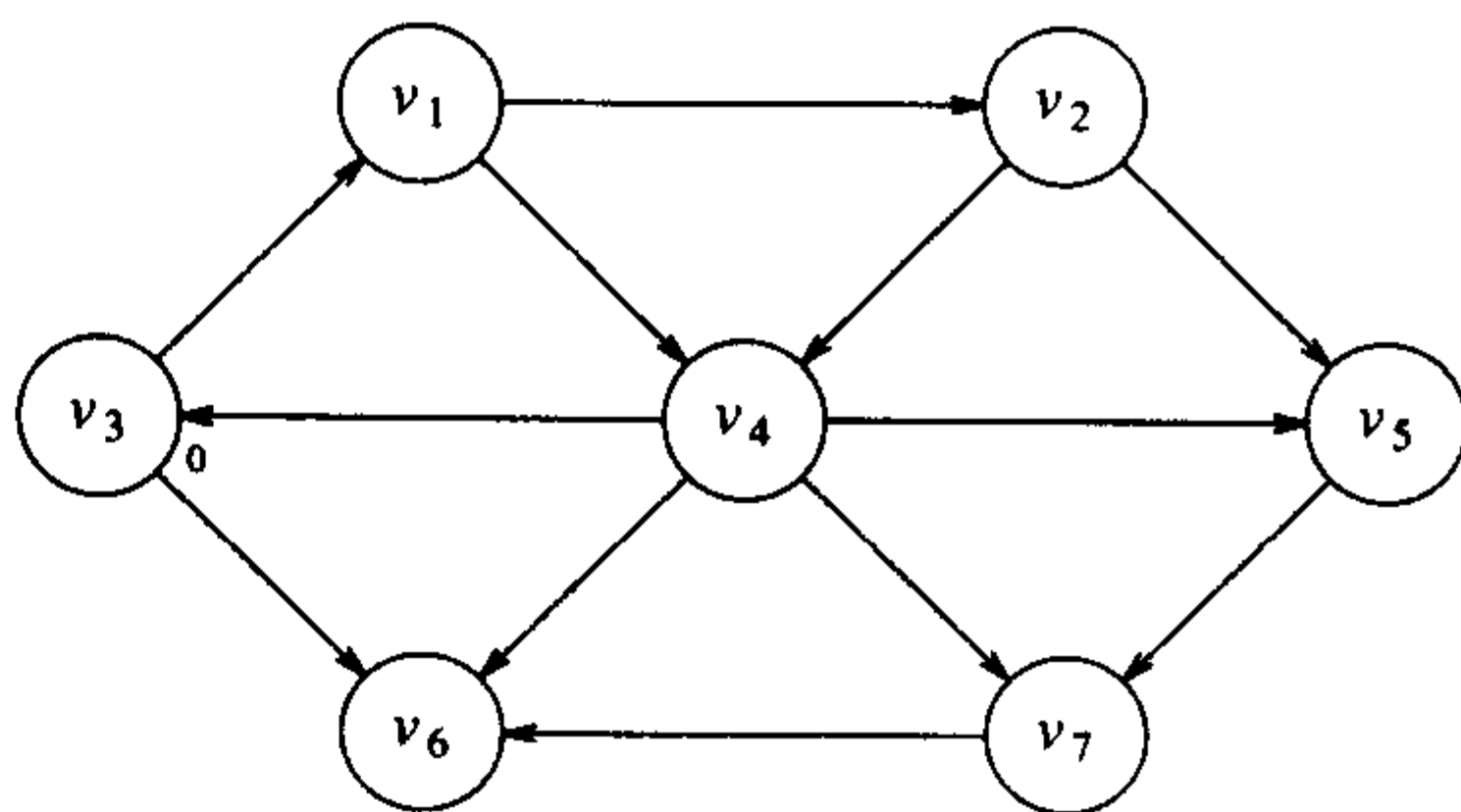


图9-11 将开始结点标记为通过0条边可以到达的结点后的图

现在可以开始寻找所有从 $s$ 出发路径长为1的顶点。这些顶点可以通过考察与 $s$ 邻接的顶点找到。此时我们看到， $v_1$ 和 $v_6$ 与 $s$ 只有一边之遥，把它们表示在图9-12中。

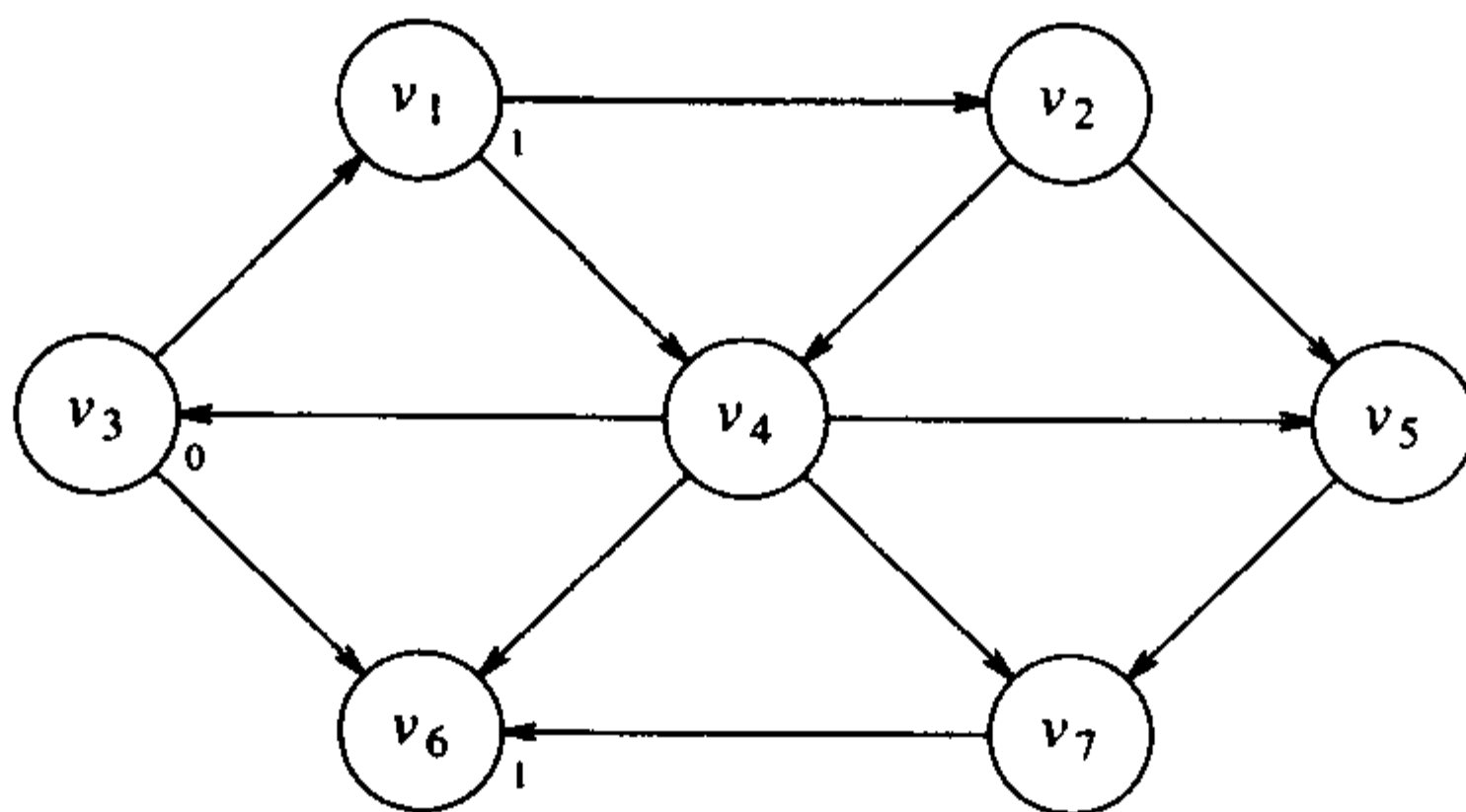


图9-12 找出所有从 $s$ 出发路径长为1的顶点之后的图

现在可以开始找出从 $s$ 出发最短路径恰为2的顶点，找出所有邻接到 $v_1$ 和 $v_6$ 的顶点（距离为1

处的顶点), 它们的最短路径还不知道。这次搜索告诉我们, 到 $v_2$ 和 $v_4$ 的最短路径长为2。图9-13显示了到现在为止所做的工作。

最后, 通过考察那些邻接到刚被赋值的 $v_2$ 和 $v_4$ 的顶点可以发现,  $v_5$ 和 $v_7$ 各有一条三边的最短路径。现在所有的顶点都已经被计算, 图9-14显示了算法的最后结果。

348

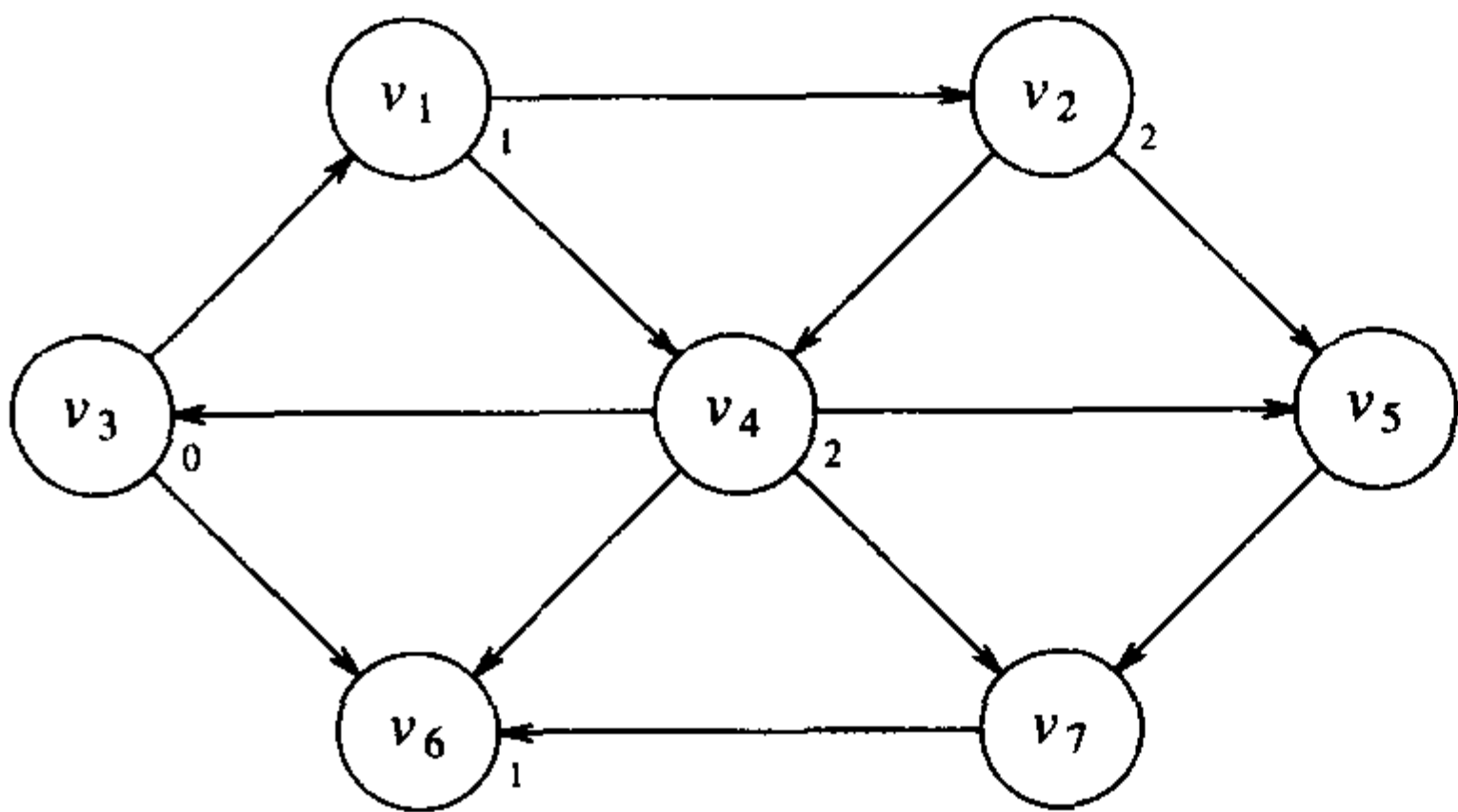


图9-13 找出所有从 $s$ 出发路径长为2的顶点之后的图

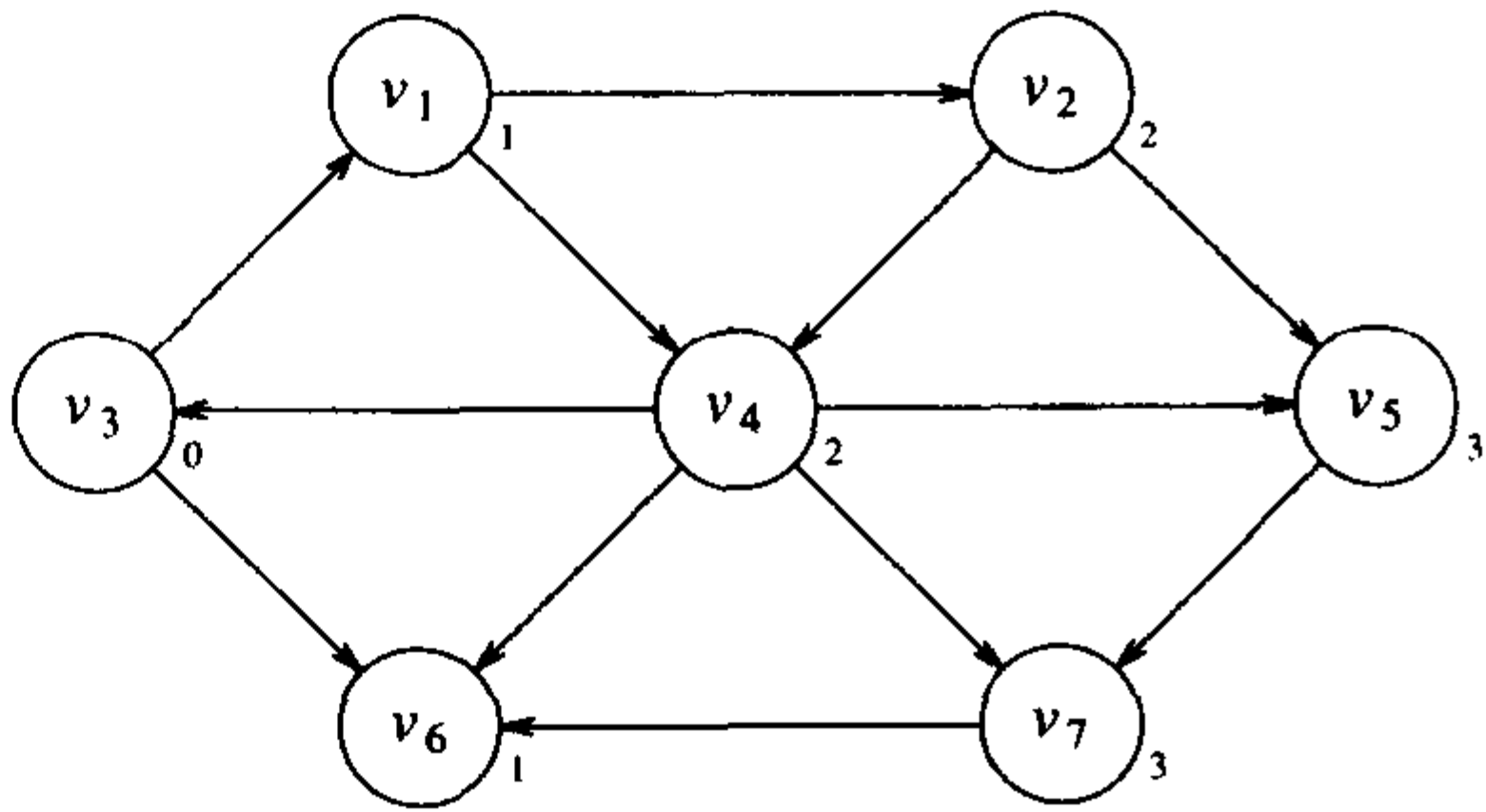


图9-14 最后的最短路径

这种搜索图的方法称为**广度优先搜索** (breadth-first search)。该方法按层处理顶点: 距开始点最近的那些顶点首先被求值, 而最远的那些顶点最后被求值。这很像对树的**层序遍历** (level-order traversal)。

有了这种方法, 我们必须把它翻译成代码。图9-15显示了该算法将要用来记录过程的表的初始配置。

| $v$   | $known$ | $d_v$    | $p_v$ |
|-------|---------|----------|-------|
| $v_1$ | F       | $\infty$ | 0     |
| $v_2$ | F       | $\infty$ | 0     |
| $v_3$ | F       | 0        | 0     |
| $v_4$ | F       | $\infty$ | 0     |
| $v_5$ | F       | $\infty$ | 0     |
| $v_6$ | F       | $\infty$ | 0     |
| $v_7$ | F       | $\infty$ | 0     |

图9-15 用于无权最短路径计算的表的初始配置

对于每个顶点, 我们将跟踪三个信息。首先, 把从 $s$ 开始到顶点的距离放到 $d_v$ 栏中。开始的时候, 除 $s$ 外所有的顶点都是不可达到的, 而 $s$ 的路径长为0。 $p_v$ 栏中的项为簿记变量, 它将使我们能够显示出实际的路径。 $known$ 中的项在顶点被处理以后置为 $true$ 。起初, 所有的顶点都不是 $known$  (已知的), 包括开始顶点。当一个顶点被标记为 $known$ 时, 我们就确信不会再找到更便宜的路径了, 因此对该顶点的处理实质上已经完成。

基本的算法在图9-16中描述。图9-16中的算法模拟这些图表, 它把距离 $d = 0$ 上的顶点声明为 $known$ , 然后声明 $d = 1$ 上的顶点为 $known$ , 再声明 $d = 2$ 上的顶点为 $known$ , 依此类推, 并且将仍然是 $d_w = \infty$ 的所有邻接的顶点 $w$ 置为距离  $d_w = d + 1$ 。

349

通过追溯 $p_v$ 变量, 可以显示实际的路径。当讨论加权的情形时, 我们将会看到如何进行。

350

由于双层嵌套for循环, 因此该算法的运行时间为 $O(|V|^2)$ 。一个明显的低效在于, 尽管所有的顶点早就成为 $known$ 了, 但是外层循环还是要继续, 直到 $NUM\_VERTICES-1$ 为止。虽然额外的附加测试可以避免这种情形发生, 但是它并不能影响最坏情形运行时间, 在以从顶点 $v_9$ 开始的图 (见图9-17) 作为输入时, 通过将所发生的情况推广即可看到这一点。



```

void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}

```

图9-16 无权最短路径算法的伪代码

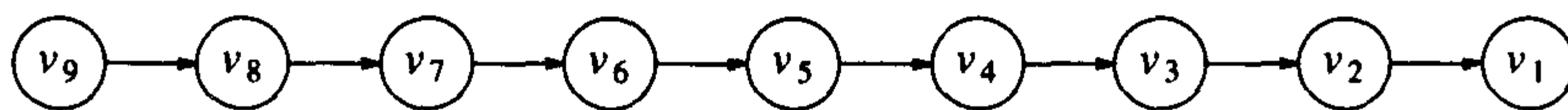


图9-17 使用图9-16（伪代码）的无权最短路径算法的坏情形

可以用非常类似于拓扑排序的做法来排除这种低效性。在任一时刻，只存在两种类型的 *unknown* 顶点，它们的  $d_v \neq \infty$ 。一些顶点的  $d_v = \text{currDist}$ ，而其余顶点的  $d_v = \text{currDist} + 1$ 。由于这种附加的结构，搜索整个的表以找出合适顶点的做法是非常浪费的。

一种非常简单但抽象的解决方案是保留两个盒子。1#盒将装有  $d_v = \text{currDist}$  的未知顶点，而2#盒则装有  $d_v = \text{currDist} + 1$  的顶点。查找适当的顶点  $v$  的测试可以用查找1#盒内的任意顶点代替。在更新  $w$ （最内层 *if* 语句的内部）以后，可以把  $w$  加到2#盒中。在最外层 *for* 循环终止以后，1#盒是空的，而2#盒则可转换成1#盒以进行下一趟 *for* 循环。

甚至可以只使用一个队列就把这种想法进一步精化。在迭代开始的时候，队列只含有距离为  $\text{currDist}$  的顶点。当添加距离为  $\text{currDist} + 1$  的邻接顶点时，由于它们自队尾入队，因此这就保证它们直到所有距离为  $\text{currDist}$  的顶点都被处理之后才被处理。在距离为  $\text{currDist}$  处的最后一个顶点出队并被处理之后，队列只含有距离为  $\text{currDist} + 1$  的顶点，因此该过程将不断进行下去。只需要把开始的结点放入队列中以启动这个过程即可。

精练的算法在图9-18中给出。在伪代码中，假设开始顶点  $s$  是作为参数被传递的。再有，如果某些顶点从开始结点出发是不可到达的，那么有可能队列会过早地变空。在这种情况下，将对这些结点报出 *INFINITY*（无穷）距离，这就完全合理了。最后，*known* 数据成员没有使用；一个顶点一旦被处理它就从不再进入队列，因此它不需要重新处理的事实就意味着被做了标记。这样一来，*known* 数据成员可以去掉。图9-19指出图上的值在算法期间是如何变化的（如果保留 *known* 数据成员，那么算法就会包含对 *known* 发生的变化）。

使用与拓扑排序同样的分析，可以看到，只要使用邻接表，运行时间就是  $O(|E| + |V|)$ 。

```
void Graph::unweighted( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue( w );
            }
    }
}
```

图9-18 无权最短路径算法的伪代码

| v              | 初始状态                            |                |                | v <sub>3</sub> 出队               |                |                | v <sub>1</sub> 出队                                |                |                | v <sub>6</sub> 出队               |                |                |
|----------------|---------------------------------|----------------|----------------|---------------------------------|----------------|----------------|--------------------------------------------------|----------------|----------------|---------------------------------|----------------|----------------|
|                | known                           | d <sub>v</sub> | p <sub>v</sub> | known                           | d <sub>v</sub> | p <sub>v</sub> | known                                            | d <sub>v</sub> | p <sub>v</sub> | known                           | d <sub>v</sub> | p <sub>v</sub> |
| v <sub>1</sub> | F                               | ∞              | 0              | F                               | 1              | v <sub>3</sub> | T                                                | 1              | v <sub>3</sub> | T                               | 1              | v <sub>3</sub> |
| v <sub>2</sub> | F                               | ∞              | 0              | F                               | ∞              | 0              | F                                                | 2              | v <sub>1</sub> | F                               | 2              | v <sub>1</sub> |
| v <sub>3</sub> | F                               | 0              | 0              | T                               | 0              | 0              | T                                                | 0              | 0              | T                               | 0              | 0              |
| v <sub>4</sub> | F                               | ∞              | 0              | F                               | ∞              | 0              | F                                                | 2              | v <sub>1</sub> | F                               | 2              | v <sub>1</sub> |
| v <sub>5</sub> | F                               | ∞              | 0              | F                               | ∞              | 0              | F                                                | ∞              | 0              | F                               | ∞              | 0              |
| v <sub>6</sub> | F                               | ∞              | 0              | F                               | 1              | v <sub>3</sub> | F                                                | 1              | v <sub>3</sub> | T                               | 1              | v <sub>3</sub> |
| v <sub>7</sub> | F                               | ∞              | 0              | F                               | ∞              | 0              | F                                                | ∞              | 0              | F                               | ∞              | 0              |
| Q:             | v <sub>3</sub>                  |                |                | v <sub>1</sub> , v <sub>6</sub> |                |                | v <sub>6</sub> , v <sub>2</sub> , v <sub>4</sub> |                |                | v <sub>2</sub> , v <sub>4</sub> |                |                |
| v              | v <sub>2</sub> 出队               |                |                | v <sub>4</sub> 出队               |                |                | v <sub>5</sub> 出队                                |                |                | v <sub>7</sub> 出队               |                |                |
|                | known                           | d <sub>v</sub> | p <sub>v</sub> | known                           | d <sub>v</sub> | p <sub>v</sub> | known                                            | d <sub>v</sub> | p <sub>v</sub> | known                           | d <sub>v</sub> | p <sub>v</sub> |
| v <sub>1</sub> | T                               | 1              | v <sub>3</sub> | T                               | 1              | v <sub>3</sub> | T                                                | 1              | v <sub>3</sub> | T                               | 1              | v <sub>3</sub> |
| v <sub>2</sub> | T                               | 2              | v <sub>1</sub> | T                               | 2              | v <sub>1</sub> | T                                                | 2              | v <sub>1</sub> | T                               | 2              | v <sub>1</sub> |
| v <sub>3</sub> | T                               | 0              | 0              | T                               | 0              | 0              | T                                                | 0              | 0              | T                               | 0              | 0              |
| v <sub>4</sub> | F                               | 2              | v <sub>1</sub> | T                               | 2              | v <sub>1</sub> | T                                                | 2              | v <sub>1</sub> | T                               | 2              | v <sub>1</sub> |
| v <sub>5</sub> | F                               | 3              | v <sub>2</sub> | F                               | 3              | v <sub>2</sub> | T                                                | 3              | v <sub>2</sub> | T                               | 3              | v <sub>2</sub> |
| v <sub>6</sub> | T                               | 1              | v <sub>3</sub> | T                               | 1              | v <sub>3</sub> | T                                                | 1              | v <sub>3</sub> | T                               | 1              | v <sub>3</sub> |
| v <sub>7</sub> | F                               | ∞              | 0              | F                               | 3              | v <sub>4</sub> | F                                                | 3              | v <sub>4</sub> | T                               | 3              | v <sub>4</sub> |
| Q:             | v <sub>4</sub> , v <sub>5</sub> |                |                | v <sub>5</sub> , v <sub>7</sub> |                |                | v <sub>7</sub>                                   |                |                | 空                               |                |                |

图9-19 无权最短路径算法期间数据如何变化

9.3.2 Dijkstra算法

如果图是加权图，那么问题（明显地）就变得困难了，不过仍然可以使用来自无权情形时的想法。

351 我们保留所有与前面相同的信息。因此，每个顶点，或者标记为*known*（已知）的，或者标记为*unknown*（未知）的。像以前一样，对每一个顶点保留一个尝试性的距离 $d_v$ 。这个距离实际上是只使用一些*known*顶点作为中间顶点从*s*到*v*的最短路径的长。和以前一样，我们记录 $p_v$ ，它是引起 $d_v$ 变化的最后的顶点。

解决单源最短路径问题的一般方法叫作Dijkstra算法，这个有30年历史的解法是贪心算法（greedy algorithm）最好的例子。贪心算法一般分阶段求解问题，在每个阶段它都把出现的东西当做是最好的去处理。例如，为了用美国货币找零钱，大部分人首先数出若干25分一个的硬币（quarter），然后是若干一角币（dime）、五分币（nickel）和一分币（penny）。这种贪心算法使用最少数目的硬币找零钱。贪心算法主要的问题在于，该算法不是总能够成功。为了找还15美分的零钱，如添加12美分一个的货币则可破坏这种找零钱算法，因为此时它给出的答案（一个12分币和三个分币）不是最优的（一个角币和一个五分币）。

352 与无权最短路径算法一样，Dijkstra算法按阶段进行。在每个阶段，Dijkstra算法选择一个顶点*v*，它在所有*unknown*顶点中具有最小的 $d_v$ ，同时算法声明从*s*到*v*的最短路径是*known*的。阶段的其余部分由 $d_w$ 值的更新工作组成。

在无权的情形，若 $d_w = \infty$ 则置 $d_w = d_v + 1$ 。因此，若顶点*v*能提供一条更短路径，则我们本质上降低了 $d_w$ 的值。如果对加权的情形应用同样的逻辑，那么当 $d_w$ 的新值 $d_v + c_{v,w}$ 是一个改进的值时，就置 $d_w = d_v + c_{v,w}$ 。简言之，使用通向*w*路径上的顶点*v*是不是一个好主意由算法决定。原始的值 $d_w$ 是不用*v*的值；上面所算出的值是使用*v*（和仅仅为*known*的顶点）的最便宜的路径。

图9-20中的图是一个例子。图9-21表示初始配置，假设开始结点*s*是 $v_1$ 。第一个选择的顶点是 $v_1$ ，路径长为0。该顶点标记为*known*。既然 $v_1$ 是*known*的，那么某些项就需要调整。邻接到 $v_1$ 的顶点是 $v_2$ 和 $v_4$ ，这两个顶点的项得到调整，如图9-22所示。

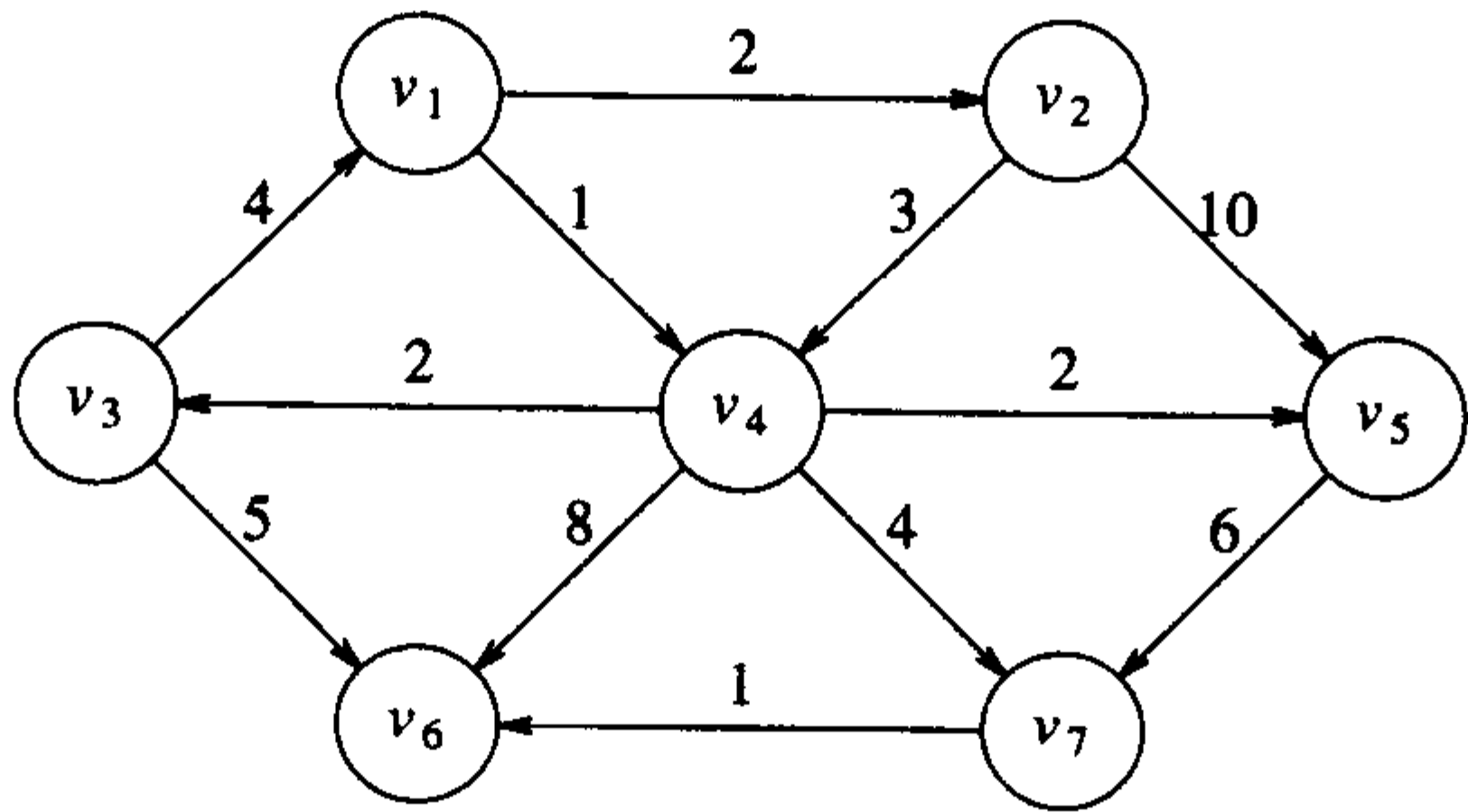


图9-20 有向图G

| <i>v</i> | <i>known</i> | $d_v$    | $p_v$ |
|----------|--------------|----------|-------|
| $v_1$    | F            | 0        | 0     |
| $v_2$    | F            | $\infty$ | 0     |
| $v_3$    | F            | $\infty$ | 0     |
| $v_4$    | F            | $\infty$ | 0     |
| $v_5$    | F            | $\infty$ | 0     |
| $v_6$    | F            | $\infty$ | 0     |
| $v_7$    | F            | $\infty$ | 0     |

图9-21 用于Dijkstra算法的表的初始配置

| <i>v</i> | <i>known</i> | $d_v$    | $p_v$ |
|----------|--------------|----------|-------|
| $v_1$    | T            | 0        | 0     |
| $v_2$    | F            | 2        | $v_1$ |
| $v_3$    | F            | $\infty$ | 0     |
| $v_4$    | F            | 1        | $v_1$ |
| $v_5$    | F            | $\infty$ | 0     |
| $v_6$    | F            | $\infty$ | 0     |
| $v_7$    | F            | $\infty$ | 0     |

图9-22 在 $v_1$ 被声明为*known*后的表

下一步，选取 $v_4$ 并标记为 $known$ 。顶点 $v_3, v_5, v_6, v_7$ 是邻接的顶点，而它们实际上都需要调整，如图9-23所示。

接下来选择 $v_2$ 。 $v_4$ 是邻接的点，但已经是 $known$ 的了，因此不需要调整。 $v_5$ 是邻接的点但不做调整，因为经过 $v_2$ 的值为 $2 + 10 = 12$ 而长为3的路径已经是已知的。图9-24显示了这些顶点被选取以后的表。

| $v$   | $known$ | $d_v$ | $p_v$ |
|-------|---------|-------|-------|
| $v_1$ | T       | 0     | 0     |
| $v_2$ | F       | 2     | $v_1$ |
| $v_3$ | F       | 3     | $v_4$ |
| $v_4$ | T       | 1     | $v_1$ |
| $v_5$ | F       | 3     | $v_4$ |
| $v_6$ | F       | 9     | $v_4$ |
| $v_7$ | F       | 5     | $v_4$ |

图9-23 在 $v_4$ 被声明为 $known$ 后

| $v$   | $known$ | $d_v$ | $p_v$ |
|-------|---------|-------|-------|
| $v_1$ | T       | 0     | 0     |
| $v_2$ | T       | 2     | $v_1$ |
| $v_3$ | F       | 3     | $v_4$ |
| $v_4$ | T       | 1     | $v_1$ |
| $v_5$ | F       | 3     | $v_4$ |
| $v_6$ | F       | 9     | $v_4$ |
| $v_7$ | F       | 5     | $v_4$ |

图9-24 在 $v_2$ 被声明为 $known$ 后

下一个被选取的顶点是 $v_5$ ，其值为3。 $v_7$ 是唯一的邻接顶点，但是它不用调整，因为 $3 + 6 > 5$ 。然后选取 $v_3$ ，对 $v_6$ 的距离下调到 $3 + 5 = 8$ 。结果如图9-25所示。

353

再下一个选取的顶点是 $v_7$ ； $v_6$ 下调到 $5 + 1 = 6$ 。得到图9-26所示的表。

| $v$   | $known$ | $d_v$ | $p_v$ |
|-------|---------|-------|-------|
| $v_1$ | T       | 0     | 0     |
| $v_2$ | T       | 2     | $v_1$ |
| $v_3$ | T       | 3     | $v_4$ |
| $v_4$ | T       | 1     | $v_1$ |
| $v_5$ | T       | 3     | $v_4$ |
| $v_6$ | F       | 8     | $v_3$ |
| $v_7$ | F       | 5     | $v_4$ |

图9-25 在 $v_5$ 及 $v_3$ 被声明为 $known$ 后

| $v$   | $known$ | $d_v$ | $p_v$ |
|-------|---------|-------|-------|
| $v_1$ | T       | 0     | 0     |
| $v_2$ | T       | 2     | $v_1$ |
| $v_3$ | T       | 3     | $v_4$ |
| $v_4$ | T       | 1     | $v_1$ |
| $v_5$ | T       | 3     | $v_4$ |
| $v_6$ | F       | 6     | $v_7$ |
| $v_7$ | T       | 5     | $v_4$ |

图9-26 在 $v_7$ 被声明为 $known$ 后

最后，我们选择 $v_6$ 。最后的表如图9-27所示。图9-28演示了在Dijkstra算法期间各边是如何标记为 $known$ 的，以及顶点是如何更新的。

为了显示出从开始顶点到某个顶点 $v$ 的实际路径，可以编写一个递归例程跟踪 $p$ 变量留下的踪迹。

| $v$   | $known$ | $d_v$ | $p_v$ |
|-------|---------|-------|-------|
| $v_1$ | T       | 0     | 0     |
| $v_2$ | T       | 2     | $v_1$ |
| $v_3$ | T       | 3     | $v_4$ |
| $v_4$ | T       | 1     | $v_1$ |
| $v_5$ | T       | 3     | $v_4$ |
| $v_6$ | T       | 6     | $v_7$ |
| $v_7$ | T       | 5     | $v_4$ |

图9-27 在 $v_6$ 被声明为 $known$ 之后，算法终止



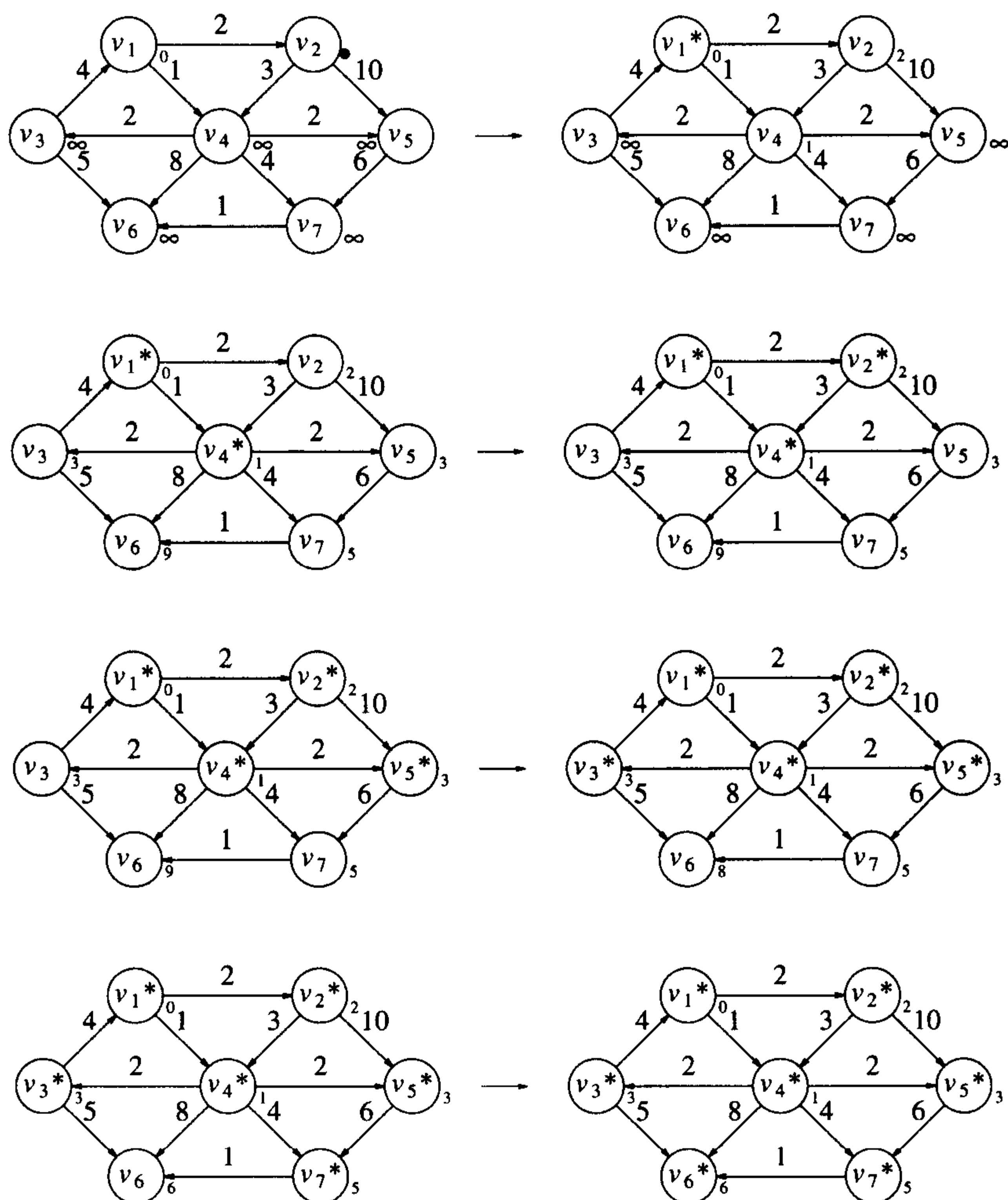


图9-28 Dijkstra算法的各个阶段

下面给出实现Dijkstra算法的伪代码。每个vertex存储在算法中使用的各种数据字段，如图9-29所示。

```

/**
 * PSEUDOCODE sketch of the Vertex structure.
 * In real C++, path would be of type Vertex *,
 * and many of the code fragments that we describe
 * require either a dereferencing * or use the
 * -> operator instead of the . operator.
 * Needless to say, this obscures the basic algorithmic ideas.
 */
struct Vertex
{
    List      adj;      // Adjacency list
    bool      known;
    DistType  dist;     // DistType is probably int
    Vertex    path;     // Probably Vertex *, as mentioned above
    // Other data and member functions as needed
};

```

图9-29 Dijkstra算法中的vertex类（伪代码）

利用图9-30中的递归例程可以显示出这个路径。该例程递归地显示路径上直到顶点 $v$ 前面的顶点的整个路径，然后再显示顶点 $v$ 。这是没有问题的，因为路径是简单的。

```
/**
 * Print shortest path to v after dijkstra has run.
 * Assume that the path exists.
 */
void Graph::printPath( Vertex v )
{
    if( v.path != NOT_A_VERTEX )
    {
        printPath( v.path );
        cout << " to ";
    }
    cout << v;
}
```

图9-30 显示实际最短路径的例程

图9-31列出主要的算法，它就是一个使用贪心选取法则填充表的for循环。

```
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
    }
}
```

图9-31 Dijkstra算法的伪代码

利用反证法证明将指出，只要所有边的值都不为负，该算法总能够顺利工作。如果任何边出现负值，则算法可能得出错误的答案（见练习9.7a）。运行时间依赖于对顶点的处理方法，这是我们必须考虑的。如果使用扫描顶点的数组来找出最小值 $d_v$ ，这种显而易见的算法，那么每一步将花费 $O(|V|)$ 时间找到最小值，从而整个算法过程中将花费 $O(|V|^2)$ 时间查找最小值。每次更新 $d_w$ 的时间是常数，而每条边最多有一次更新，总计为 $O(E)$ 。因此，总的运行时间为 $O(|E| + |V|^2) = O(|V|^2)$ 。如果图是稠密的，边数 $|E| = \Theta(|V|^2)$ ，则该算法不仅简单而且基本上最优，因为其运行时间与边数

357 成线性关系。

如果图是稀疏的，边数 $|E| = \Theta(|V|)$ ，那么这种算法就太慢了。在这种情况下，距离需要存储在优先队列中。有两种方法可以做到这一点，二者是类似的。

顶点 $v$ 的选择是一个`deleteMin`操作，因为一旦未知的最小值顶点被找到，那么它就不再是未知的，以后就不必再考虑。 $w$ 的距离的更新可以有两种方法实现。

358 一种方法是把更新看成`decreaseKey`操作。此时，查找最小值的时间为 $O(\log |V|)$ ，即执行更新的时间，它相当于执行`decreaseKey`操作的时间。由此得出运行时间为 $O(|E|\log |V| + |V|\log |V|) = O(|E|\log |V|)$ ，它是对前面稀疏图的界的改进。由于优先队列不能有效地支持`find`操作，因此 $d_i$ 的每个值在优先队列中的位置将需要保留，并当 $d_i$ 在优先队列中改变时随之更新。如果优先队列是用二叉堆实现的，那么这将很凌乱。如果使用配对堆（pairing heap，见第12章），则程序不会太差。

另一种方法是在每次 $w$ 的距离变化时，就把 $w$ 和新值 $d_w$ 插入到优先队列中去。这样，对优先队列中的每个顶点就可能有多个表示。当`deleteMin`操作把最小的顶点从优先队列中删除时，必须检查以确保它不是`known`的。如果是`known`的，就简单地将其忽略，执行下一个`deleteMin`。这种方法虽然从软件的观点看是优越的，而且编程确实容易得多，但是，优先队列的大小可能达到 $|E|$ 这么大。由于 $|E| \leq |V|^2$ 意味着 $\log |E| \leq 2\log |V|$ ，因此这并不影响渐近时间界。这样，我们仍然得到一个 $O(|E|\log |V|)$ 算法。不过，空间需求的确增加了，在某些应用中这可能是严重的。不仅如此，因为该方法需要 $|E|$ 次而不是仅仅 $|V|$ 次`deleteMin`，所以它在实践中很可能更慢。

359 注意，对于一些诸如计算机邮件和大型公交运输的典型问题，它们的图一般是非常稀疏的，因为大多数顶点只有少数几条边。因此，在许多应用中使用优先队列来解决这种问题是很重要的。

如果使用不同的数据结构，那么Dijkstra算法可能会有更好的时间界。在第11章，我们将看到另外的优先队列数据结构，叫作斐波那契堆，使用这种数据结构的运行时间是 $O(|E| + |V|\log |V|)$ 。斐波那契堆具有良好的理论时间界，不过，它需要相当数量的系统开销。因此，尚不清楚在实践中使用斐波那契堆是否比使用带有二叉堆的Dijkstra算法更好。至今，这个问题尚没有有意义的平均情形的结果。

如果使用不同的数据结构，那么Dijkstra算法可能会有更好的时间界。在第11章，我们将看到另外的优先队列数据结构，叫作斐波那契堆，使用这种数据结构的运行时间是 $O(|E| + |V|\log |V|)$ 。斐波那契堆具有良好的理论时间界，不过，它需要相当数量的系统开销。因此，尚不清楚在实践中使用斐波那契堆是否比使用带有二叉堆的Dijkstra算法更好。至今，这个问题尚没有有意义的平均情形的结果。

### 9.3.3 具有负边值的图

如果图具有负的边值，那么Dijkstra算法是行不通的。问题在于，一旦一个顶点 $u$ 被声明为`known`，那就可能从某个另外的`unknown`顶点 $v$ 有一条回到 $u$ 的负的路径。在这样的情形下，选取从 $s$ 到 $v$ 再回到 $u$ 的路径要比选取从 $s$ 到 $u$ 但不使用 $v$ 的路径更好。练习9.7a要求构造一个明晰的例子。

一个诱人的方案是将一个常数 $\Delta$ 加到每一条边的值上，如此除去负的边，再计算新图的最短路径问题，然后把结果用到原来的图上。这种方案的直接实现是行不通的，因为那些具有许多条边的路径变成比那些具有很少边的路径权重更重了。

把加权的和无权的算法结合起来可以解决这个问题，但是要付出运行时间急剧增长的代价。我们忽略了关于`known`顶点的概念，因为算法需要能够改变它的意向。开始，我们把 $s$ 放到队列中。然后，在每一阶段让一个顶点 $v$ 出队。找出所有与 $v$ 邻接的顶点 $w$ ，使得 $d_w > d_v + c_{v,w}$ 。然后更新 $d_w$ 和 $p_w$ ，并在 $w$ 不在队列中时把它放到队列中。可以为每个顶点设置一个位（bit）以指示它在队列中出现的情况。重复这个过程直到队列为空。图9-32（几乎）实现了这个算法。

```

void Graph::weightedNegative( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( v.dist + cvw < w.dist )
            {
                // Update w
                w.dist = v.dist + cvw;
                w.path = v;
                if( w is not already in q )
                    q.enqueue( w );
            }
    }
}

```

图9-32 具有负边值的加权最短路径算法的伪代码

虽然如果没有负值回路该算法能够正常工作，但是，内层的for循环的代码每边只执行一次的情况不再成立。每个顶点最多可以出队 $|V|$ 次，因此，如果使用邻接表则运行时间是 $O(|E| \cdot |V|)$ （练习9.7b）。这比Dijkstra算法多很多，幸运的是，实践中边的值是非负的。如果负值回路存在，那么算法将无限地循环下去。通过在任一顶点已经出队 $|V|+1$ 次后停止算法运行，可以保证它能终止。

### 9.3.4 无环图

如果知道图是无环的，那么可以通过改变声明顶点为*known*的顺序，或者叫作顶点选取法则，来改进Dijkstra算法。新法则是以拓扑顺序选择顶点。由于选择和更新可以在拓扑排序执行的时候进行，因此算法能够一趟完成。

360

因为当一个顶点 $v$ 被选取以后，按照拓扑排序的法则它没有从*unknown*顶点发出的进入边，因此它的距离 $d_v$ 可以不再降低，所以这种选取法则是行得通的。

使用这种选取法则不需要优先队列；由于选取花费常数时间，因此运行时间为 $O(|E|+|V|)$ 。

无环图可以模拟某种下坡滑雪问题——我们想要从点 $a$ 到点 $b$ ，但只能走下坡，显然不可能有回路。另一个可能的应用是（不可逆的）化学反应模型。可以让每个顶点代表实验的一个特定的状态，让边代表从一种状态到另一种状态的转变，而边的权代表释放的能量。如果只能从高能状态转变到低能状态，那么图就是无环的。

无环图的一个更重要的用途是**关键路径分析**（critical path analysis）。我们将用图9-33作为例子。每个结点表示一个必须执行的动作以及完成动作所花费的时间。因此，该图叫作**动作结点图**（activity-node graph）。图中的边代表优先关系：一条边 $(v, w)$ 意味着动作 $v$ 必须在动作 $w$ 开始前完成。当然，这就意味着图必须是无环的。假设任何（直接或间接）互相不依赖的动作可以由不同的服务器并行地执行。

361



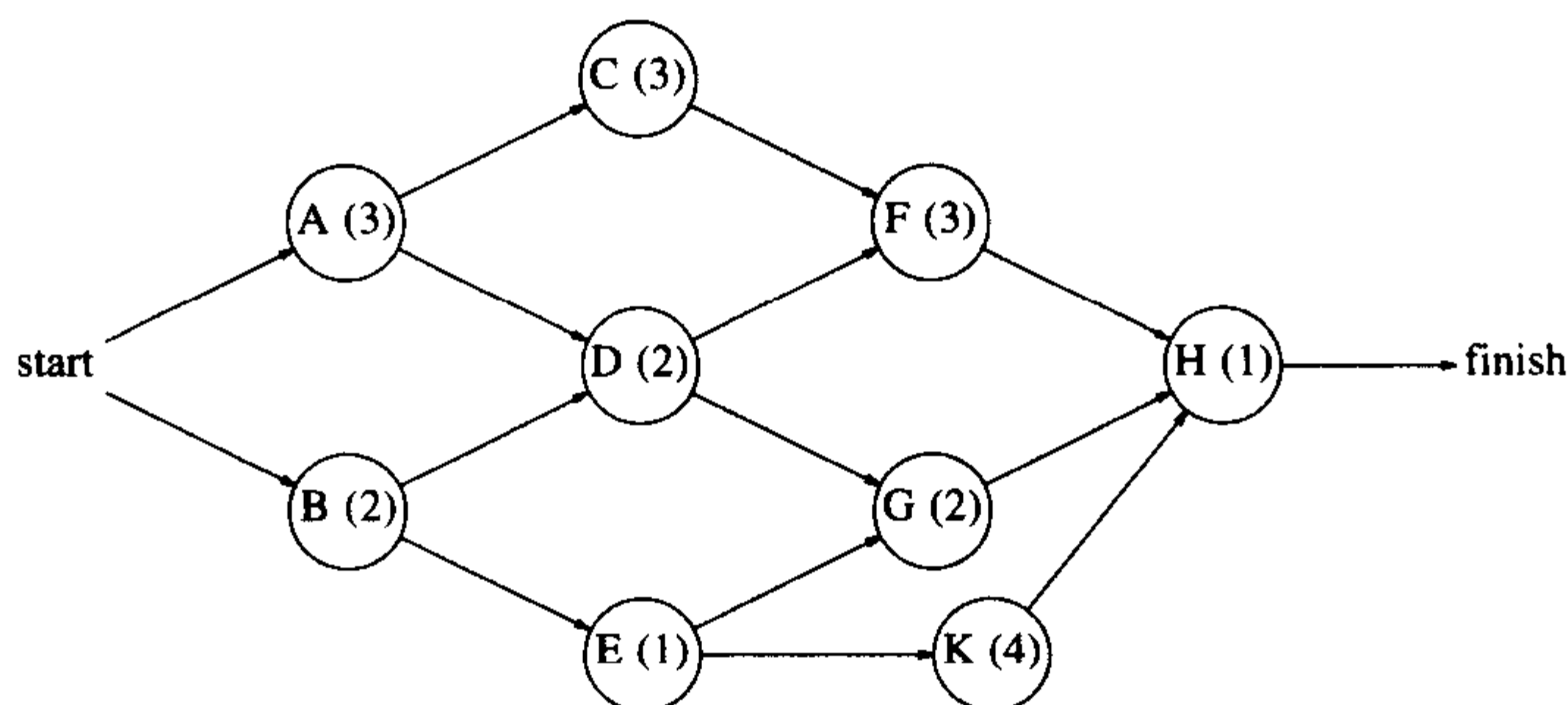


图9-33 动作结点图

这种类型的图可以（并常常）用来模拟方案的构建。在这种情况下，有几个重要的问题需要回答。首先，方案的最早完成时间是何时？从图中可以看到，沿路径A、C、F、H需要10个时间单位。另一个重要的问题是确定哪些动作可以延迟，延迟多长，而不至于影响最少完成时间。例如，延迟A、C、F、H中的任一个都将使完成时间超过10个时间单位。另一方面，动作B不是很关键，可以被延迟两个时间单位而不会影响最后完成时间。

为了进行这些运算，我们把动作结点图转化成事件结点图（event-node graph）。每个事件对应一个动作和所有与它相关的动作的完成。从事件结点图中的结点 $v$ 可达到的事件可以在事件 $v$ 完成后开始。这个图可以自动构造，也可以人工构造。在一个动作依赖于几个其他动作的情况下，可能需要插入哑边和哑结点。为了避免引进假相关性（或相关性的假短缺），这么做是必要的。对应图9-33的事件结点图如图9-34所示。

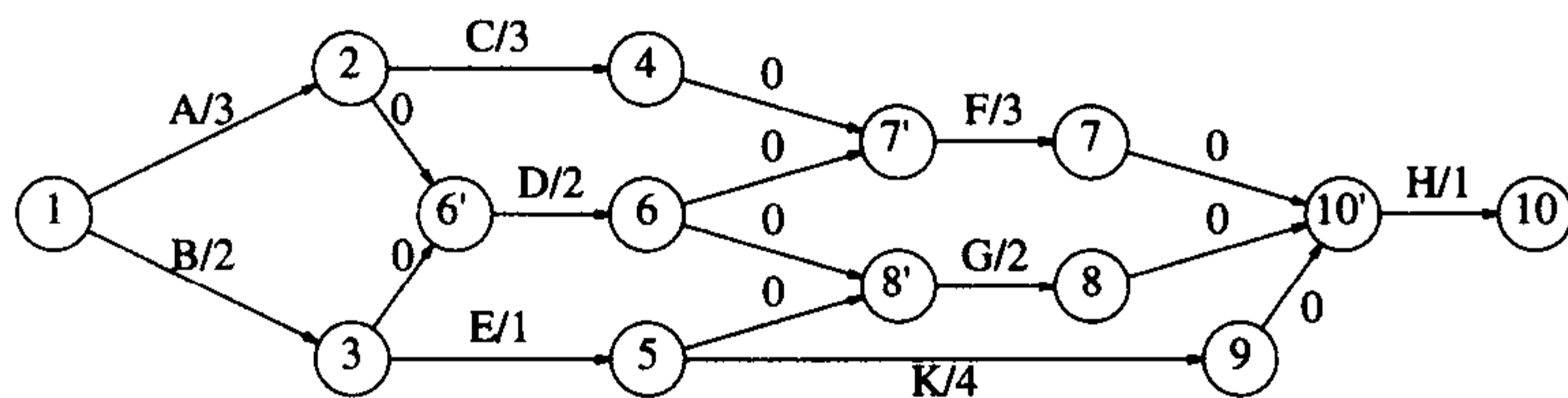


图9-34 事件结点图

为了找出方案的最早完成时间，只要找出从第一个事件到最后一个事件的最长路径的长。对于一般的图，最长路径问题通常没有意义，因为可能有正值回路（positive-cost cycle）存在。这些正值回路等价于最短路径问题中的负值回路。如果出现正值回路，那么可以寻找最长的简单路径，不过，对于这个问题还没有已知的圆满的解决方案。由于事件结点图是无环图，因此不需要关心回路的问题。在这种情况下，采纳最短路径算法计算图中所有结点的最早完成时间是很容易的。如果 $EC_i$ 是结点 $i$ 的最早完成时间，那么可用的法则为：

362

$$EC_1 = 0$$

$$EC_w = \max_{(v,w) \in E} (EC_v + c_{v,w})$$

图9-35显示了该例子的事件结点图中每个事件的最早完成时间。

还可以计算每个事件能够完成而不影响最后完成时间的最晚时间 $LC_i$ 。进行这项工作的公式为：

$$LC_n = EC_n$$

$$LC_v = \min_{(v,w) \in E} (LC_w - c_{v,w})$$

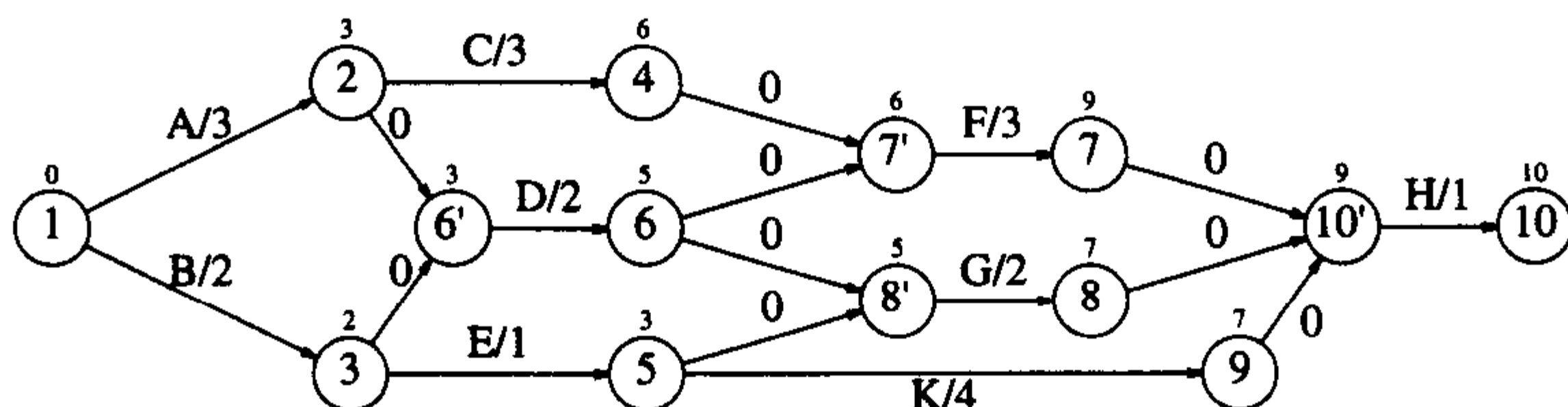


图9-35 最早完成时间

对于每个顶点，通过保存一个所有邻接且在前的顶点的表，这些值就可以以线性时间算出。借助顶点的拓扑顺序计算最早完成时间，而最晚完成时间则通过倒转拓扑顺序来计算。最晚完成时间如图9-36所示。

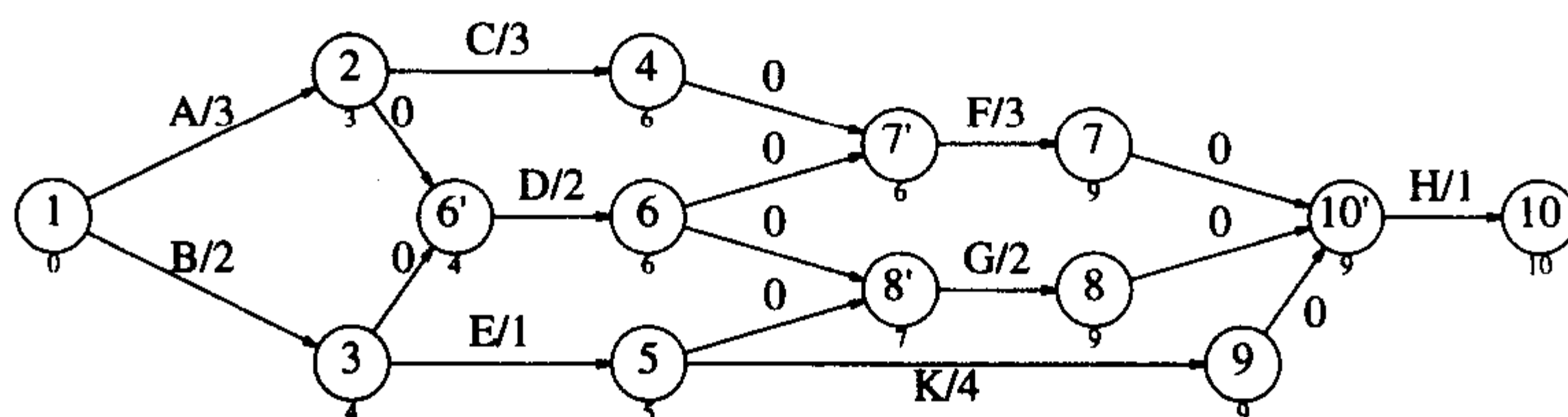


图9-36 最晚完成时间

事件结点图中每条边的松弛时间 (slack time) 代表对应动作可以被延迟而不推迟整体完成的时间量。容易看出：

$$Slack_{(v,w)} = LC_w - EC_v - c_{v,w}$$

363

图9-37显示了事件结点图中每个动作的松弛时间 (作为第三项)。对于每个结点，上面的数字是最早完成时间，下面的数字是最晚完成时间。

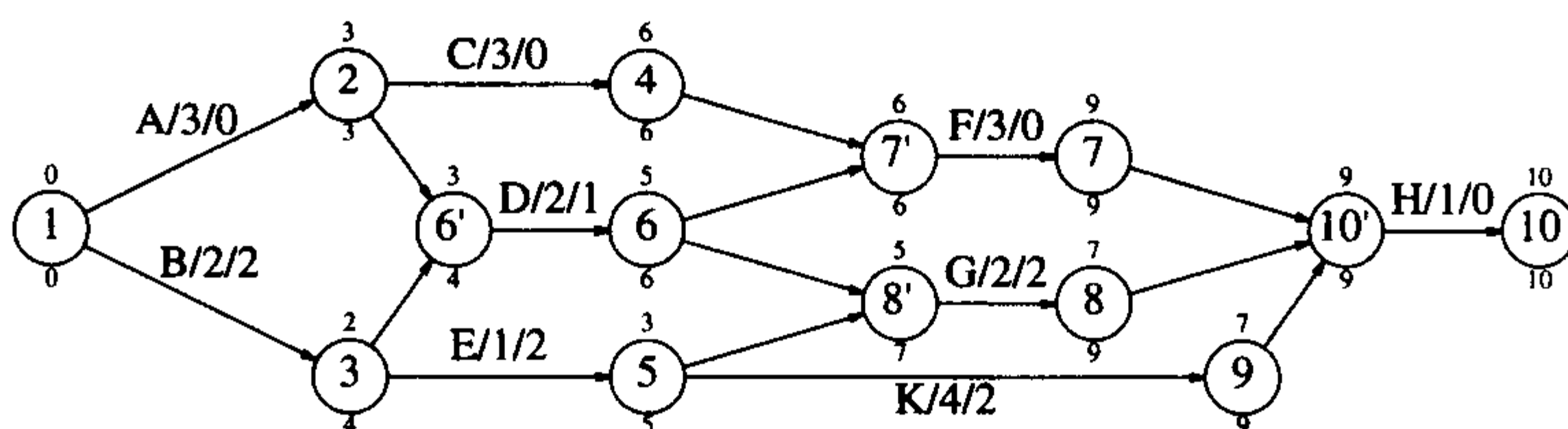


图9-37 最早完成时间、最晚完成时间和松弛时间

某些动作的松弛时间为零，这些动作是关键性的动作，它们必须按计划结束。至少存在一条完全由零-松弛边组成的路径，这样的路径是**关键路径** (critical path)。

### 9.3.5 所有顶点对的最短路径

有时重要的是要找出图中所有顶点之间的最短路径。虽然可以运行 $|V|$ 次适当的单源算法，但是如果立即计算所有的信息，我们还是期望有更快的解法，尤其是对于稠密的图。

在第10章，我们将看到对加权图求解这种问题的一个 $O(|V|^3)$ 算法。虽然对于稠密图它具有和运行 $|V|$ 次简单 (非优先队列) Dijkstra算法相同的时间界，但是它的循环是如此地紧凑以至于所有专业的顶点间算法很可能在实践中更快。当然，对于稀疏图更快的是运行 $|V|$ 次用优先队列编写的Dijkstra算法。

364

### 9.3.6 最短路径举例

本节编写一些C++例程来计算字梯。在字梯游戏中，每一个词都是通过将字梯中的前一个词改变一个字母形成的。例如，我们可以通过一系列的单字母替换将zero转换成five: zero、hero、here、hire、fire、five。

这是一个无权最短路径问题，其中每个词是一个顶点，如果两个顶点可以通过一个字母的替换相互转化的话，在这两个顶点之间就有一条（双向）边。

在4.8节中，我们描述并编写了一个C++例程，来生成一个图。其键为词，值为包含可以通过一个字母转换获得的词的vector。一样的，这里的map代表图，为邻接表形式。我们仅仅需要编写一个例程来运行单源无权最短路径算法，然后再编写第二个程序在单源最短路径算法完成的时候，输出词的序列。这两个例程都在图9-38中给出。

```

1 // Runs the shortest path calculation from the adjacency map, returning a map
2 // that contains the "prev" entries for each word in the graph.
3 map<string,string> findChain( const map<string,vector<string> > & adjacentWords,
4                               const string & first )
5 {
6     map<string,string> previousWord;
7     queue<string> q;
8
9     q.push( first );
10    while( !q.empty( ) )
11    {
12        string current = q.front( ); q.pop( );
13
14        map<string,vector<string> >::const_iterator itr;
15        itr = adjacentWords.find( current );
16
17        const vector<string> & adj = itr->second;
18        for( int i = 0; i < adj.size( ); i++ )
19            if( previousWord[ adj[ i ] ] == "" )
20            {
21                previousWord[ adj[ i ] ] = current;
22                q.push( adj[ i ] );
23            }
24    }
25    previousWord[ first ] = "";
26
27    return previousWord;
28 }
29
30 // After the shortest path calculation has run, computes the vector that
31 // contains the sequence of word changes to get from first to second.
32 vector<string> getChainFromPrevMap( const map<string,string> & previous,
33                                     const string & second )
34 {
35     vector<string> result;
36     map<string,string> & prev = const_cast<map<string,string> >&( previous );
37
38     for( string current = second; current != ""; current = prev[ current ] )
39         result.push_back( current );
40
41     reverse( result.begin( ), result.end( ) );
42
43     return result;
44 }

```

图9-38 查找字梯的C++程序

第一个例程是findChain。该例程用map代表邻接表和两个连接的词。返回一个map，其中键为词，相应的值为以first开始的最短梯子中的键前面的那个词。换言之，在上面的例子中，如果开始词为零，键five的值为fire；键fire的值为hire；键hire的值为here，等等。很明显，这给第二个例程getChainFromPreviousMap提供了足够的信息，使之可以从后向前来完成任务。

findChain是图9-18中伪代码的直接实现。为简单起见，假设first为adjacentWords的一个键（这可以在调用之前很容易地测试，或者可以在第16行添加额外的代码，在这个条件不满足时抛出一个异常）。基本循环将前一个项错误地赋给first（当邻接到first的原始词被处理后）。在第25行，该项被修正。

getChainFromPrevMap使用prev图和second（second大概为map中的一个键），通过prev从后向前执行，并返回用以形成字梯的词。由于生成的词是逆序的，所以，使用STL reverse算法来解决这个问题。第36行的类型转换是必需的，因为operator[]不能用在不可变的map。

因为这段代码使用了map，因此，运行时间是 $\log E$ 级的，比实际需要的稍高一点。该问题可以通过使用hash\_map来避免。

也可以将这个问题扩展为包含有删除一个字母或增加一个字母的单字母替换问题。要计算这个邻接表仅需要做少许附加的工作。在4.8节的算法最后，每次在组g中代表词w的变量处理的时候，我们都检查这个变量是不是组g-1中的一个词。如果是，那么这个变量就邻接到w（这是单字母删除），同时w邻接到这个代表（这是单字母增加）。也可以给字母删除或插入赋值（这比简单的替代复杂一点），这导致一个加权最短路径问题，这个问题可以使用Dijkstra算法解决。

365  
366

## 9.4 网络流问题

设给定边容量为 $c_{v,w}$ 的有向图 $G = (V, E)$ 。这些容量可以代表通过一个管道的水的流量或两个交叉路口之间马路上的交通流量。有两个顶点，一个是s，称为源点（source），一个是t，称为汇点（sink）。对于任一条边 $(v, w)$ ，最多有“流”的 $c_{v,w}$ 个单位可以通过。在既不是源点s又不是汇点t的任一顶点v，总的进入的流必须等于总的发出的流。最大流问题就是确定从s到t可以通过的最大流量。例如，对于图9-39中左边的图，最大流是5，如右边的图所示。

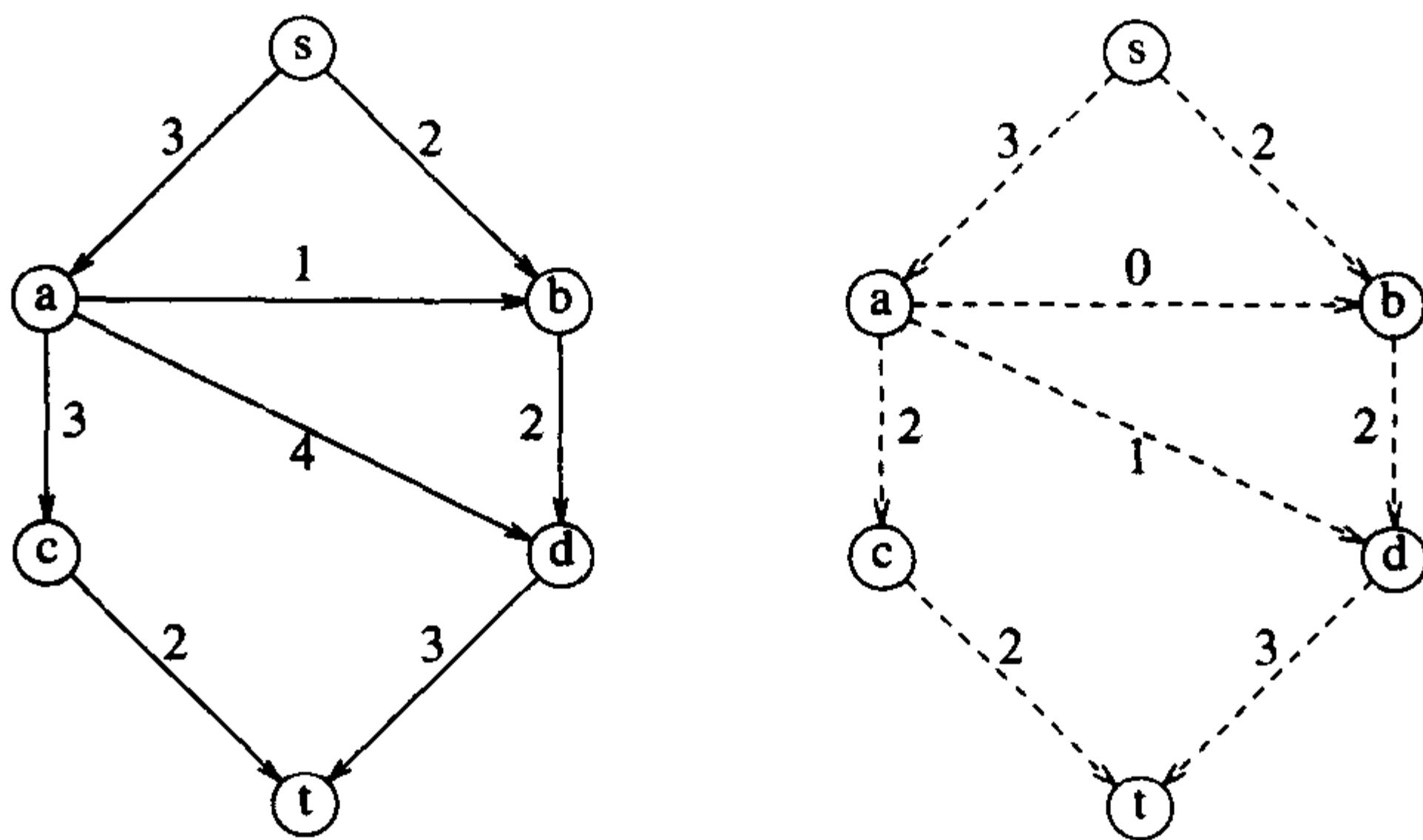


图9-39 一个图（左边）和它的最大流

正如问题叙述中所要求的，没有边负载超过它的容量的流。顶点a有3个单位的流进入，它将这3个单位的流分转给c和d。顶点d从a和b得到3个单位的流，并把它们结合起来发送到t。顶点可



以以任何方式结合和发送流，只要不违反边的容量以及保持流守恒（进入必须流出）即可。

### 一个简单的最大流算法

解决这种问题的首要想法是分阶段进行。我们从图 $G$ 开始并构造一个流图 $G_f$ 。 $G_f$ 表示在算法的任意阶段已经达到的流。开始时 $G_f$ 的所有边都没有流，我们希望当算法终止时 $G_f$ 包含最大流。我们还构造一个图 $G_r$ ，称为**残余图**（residual graph），它表示对于每条边还能再添加多少流。对于每一条边，可以从容量中减去当前的流而计算出残余的流。 $G_r$ 的边叫作**残余边**（residual edge）。

**367** 在每个阶段，我们寻找图 $G_r$ 中从 $s$ 到 $t$ 的一条路径，这条路径叫作**增长路径**（augmenting path）。这条路径上的最小边值就是可以添加到路径上每一边的流的量。我们通过调整 $G_f$ 和重新计算 $G_r$ 做到这一点。当发现 $G_r$ 中没有从 $s$ 到 $t$ 的路径时算法终止。这个算法是不确定的，因为从 $s$ 到 $t$ 的路径是任意选择的。显然，有些选择会更好，后面我们再讨论这个问题。我们将对我们的例子运行这个算法。下面的图分别是 $G$ 、 $G_f$ 和 $G_r$ 。要记住，这个算法有一个小缺陷。初始的配置见图9-40。

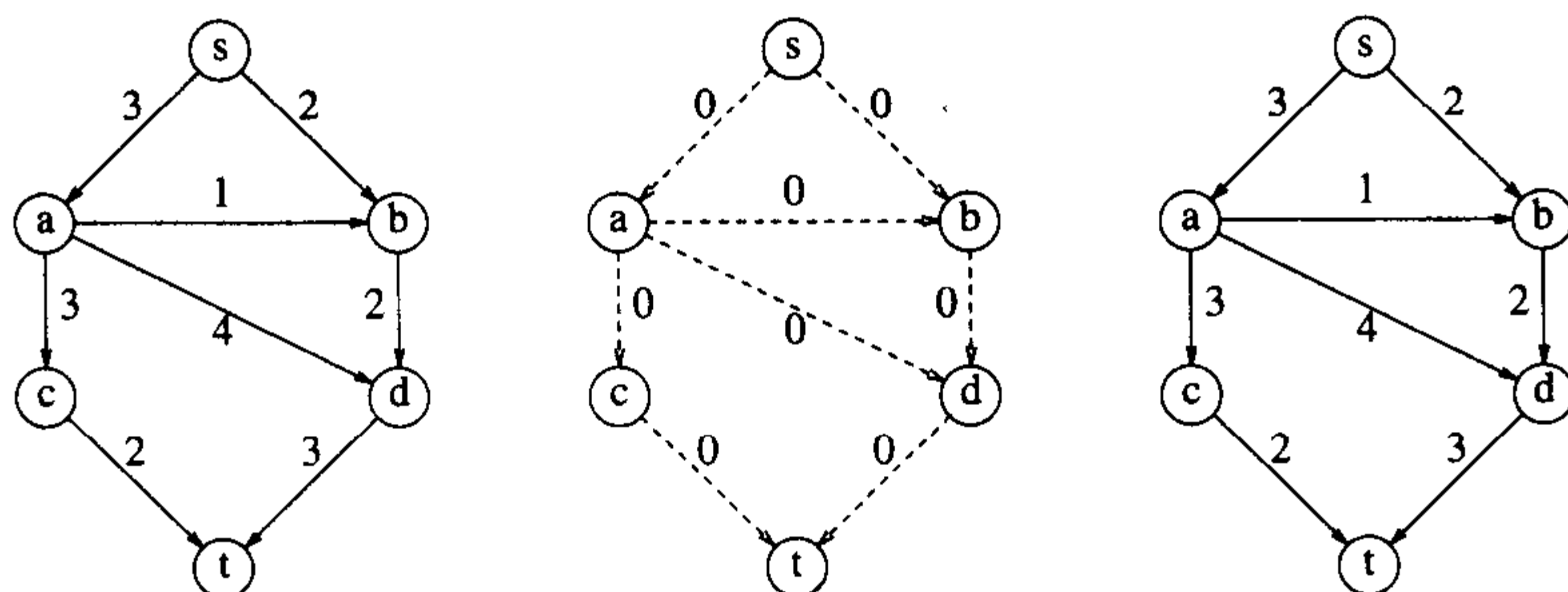


图9-40 图、流图以及残余图的初始阶段

在残余图中有许多从 $s$ 到 $t$ 的路径。假设我们选择 $s$ 、 $b$ 、 $d$ 、 $t$ 。此时可以发送2个单位的流通过这条路径的每一边。我们将采取以下约定：一旦注满（使饱和）一条边，则这条边就要从残余图中除去。这样，可以得到图9-41。

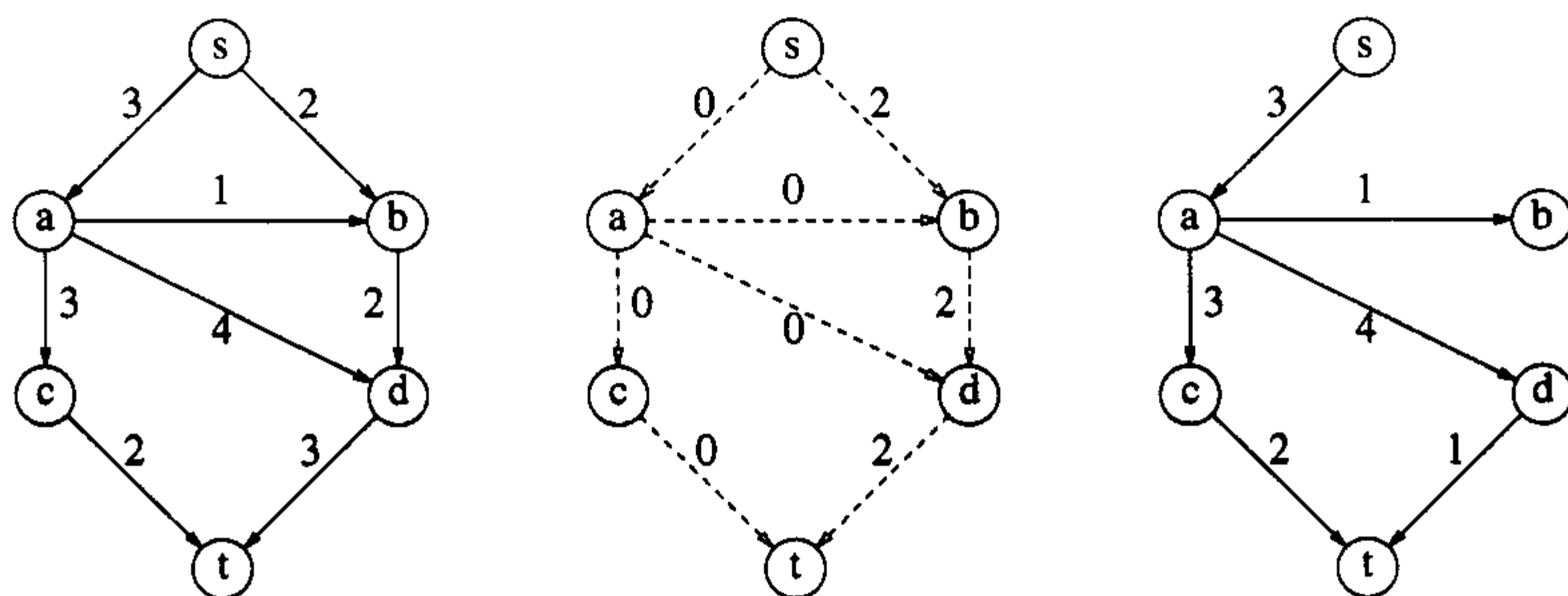
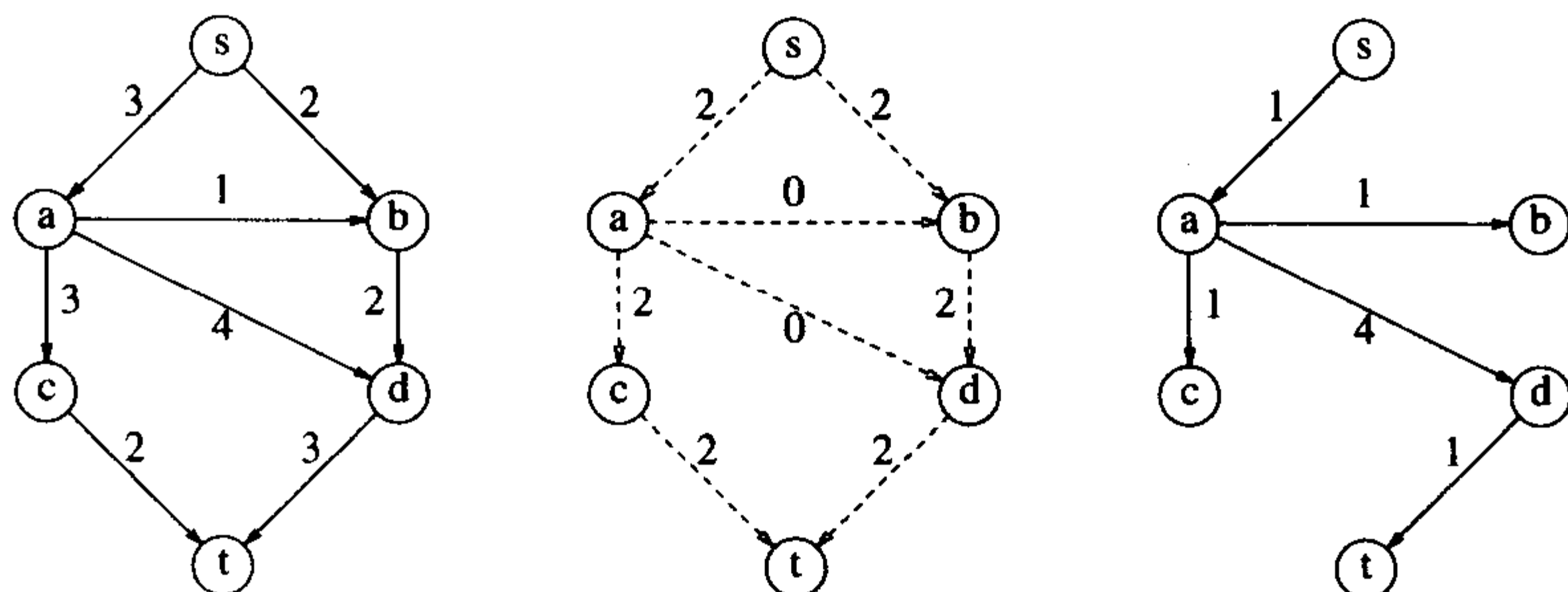
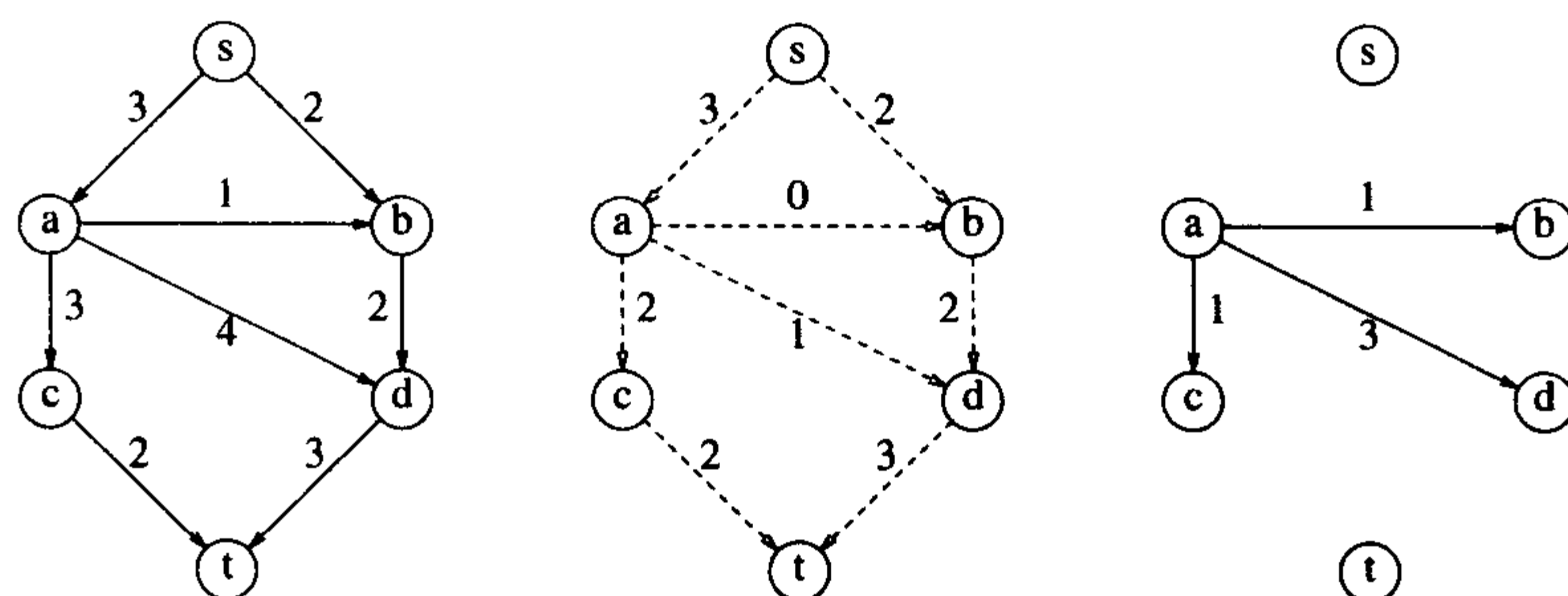


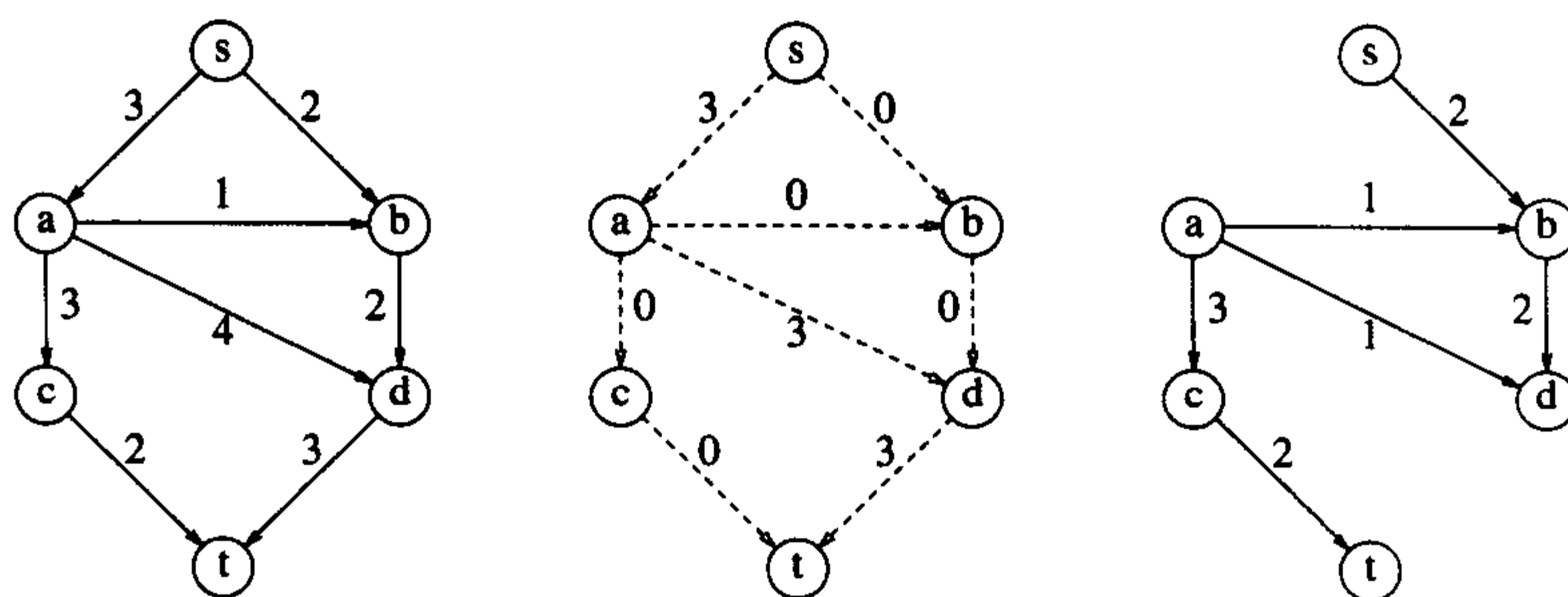
图9-41 沿 $s$ 、 $b$ 、 $d$ 、 $t$ 加入2个单位的流后的 $G$ 、 $G_f$ 、 $G_r$

下面，选择路径 $s$ 、 $a$ 、 $c$ 、 $t$ ，该路径也容许2个单位的流通过。进行必要的调整后，我们得到图9-42。

**368** 唯一剩下的路径是 $s$ 、 $a$ 、 $d$ 、 $t$ ，这条路径能够容纳1个单位的流通过。结果如图9-43所示。

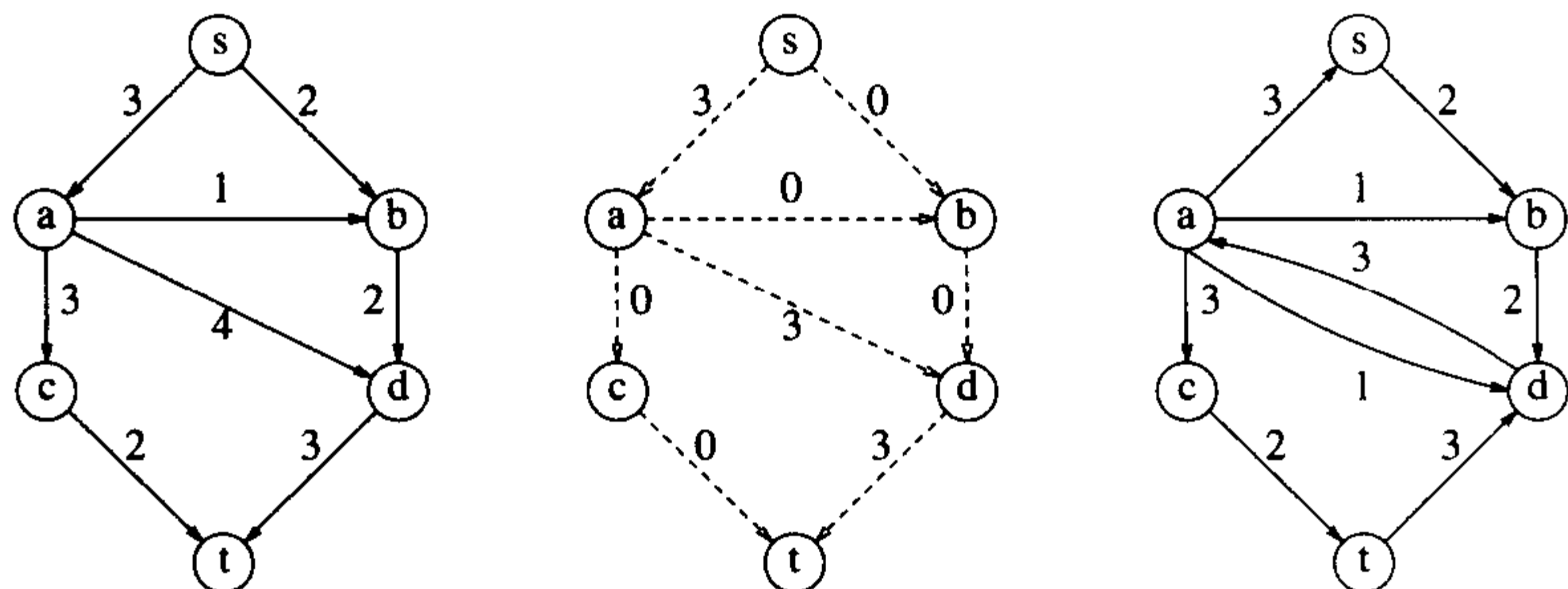
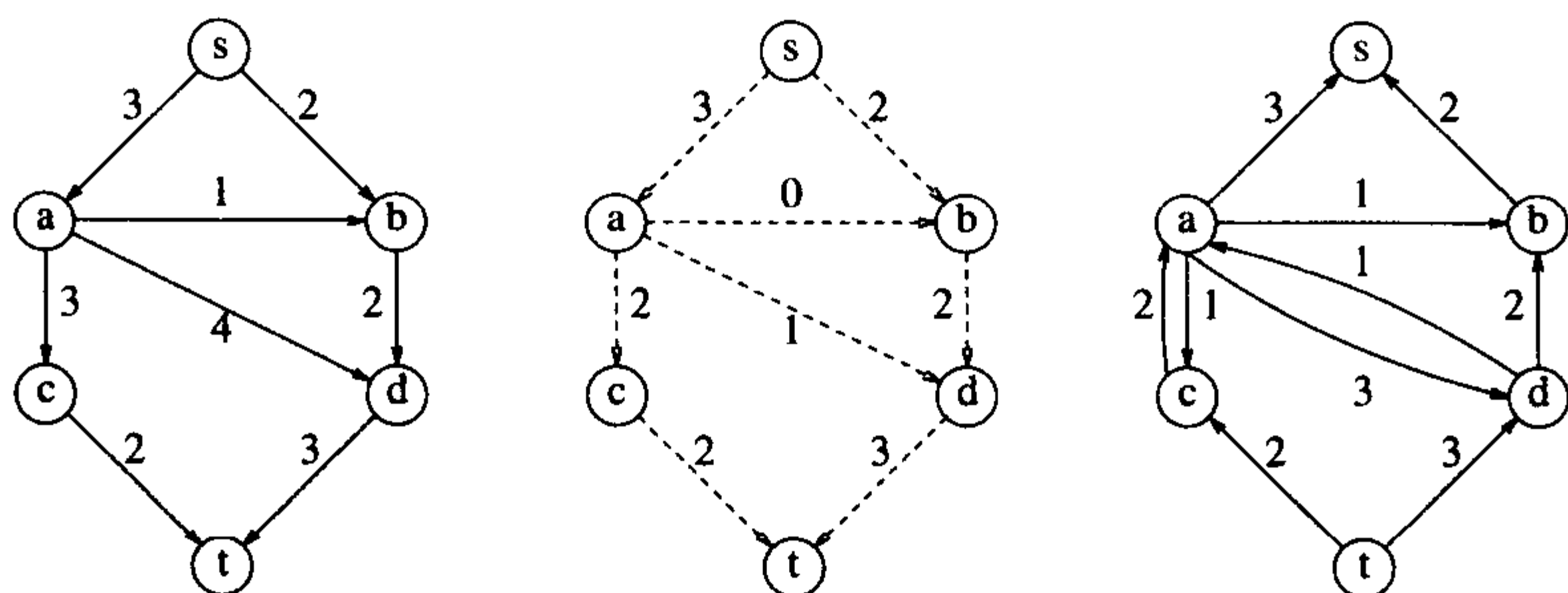
图9-42 沿 $s$ 、 $a$ 、 $c$ 、 $t$ 加入2个单位的流后的 $G$ 、 $G_f$ 、 $G_r$ 图9-43 沿 $s$ 、 $a$ 、 $d$ 、 $t$ 加入1个单位的流后的 $G$ 、 $G_f$ 、 $G_r$ ——算法终止

由于 $t$ 从 $s$ 出发是不可达到的，因此算法到此终止。结果正好5个单位的流是最大值。为了看清问题的所在，设从初始图开始我们选择路径 $s$ 、 $a$ 、 $d$ 、 $t$ ，这条路径容纳3个单位的流，因而似乎是一种好的选择。然而选择的结果却使得残余图中不再有从 $s$ 到 $t$ 的任何路径，因此，该算法不能找到最优解。这是贪心算法行不通的一个例子。图9-44指出为什么算法会失败。

图9-44 如果初始动作是沿 $s$ 、 $a$ 、 $d$ 、 $t$ 加入3个单位的流得到 $G$ 、 $G_f$ 、 $G_r$ ——算法终止但解不是最优的

为了使得算法有效，需要让算法改变它的意向。为此，对于流图中具有流 $f_{v,w}$ 的每一边 $(v, w)$ ，我们将在残余图中添加一条容量为 $f_{v,w}$ 的边 $(w, v)$ 。事实上，可以通过以相反的方向发回一个流而使算法改变它的意向。通过例子最能看清这个问题。我们从原始的图开始并选择增长路径 $s$ 、 $a$ 、 $d$ 、 $t$ ，得到图9-45。

注意，在残余图中有些边在 $a$ 和 $d$ 之间有两个方向。或者还有1个单位的流可以从 $a$ 导向 $d$ ，或者有高达3个单位的流导向相反的方向——我们可以撤销流。现在算法找到流为2的增长路径 $s$ 、 $b$ 、 $d$ 、 $a$ 、 $c$ 、 $t$ 。通过从 $d$ 到 $a$ 导入2个单位的流，算法从边 $(a, d)$ 取走2个单位的流，因此本质上改变了它的意向。图9-46给出新的图。

图9-45 使用正确的算法沿 $s, a, d, t$ 加入3个单位的流后的图图9-46 使用正确算法沿 $s, b, d, a, c, t$ 加入2个单位的流后的图

在这个图中没有增长路径，因此，算法终止。奇怪的是，可以证明，如果边的容量都是有理数，那么该算法总以最大流终止。证明多少有些困难，也超出了本书的范围。虽然例子正好是无环的，但这并不是算法有效工作所必需的。使用无环图只是为了简明起见。

如果容量都是整数且最大流为 $f$ ，那么，由于每条增长路径使流的值至少增1，故 $f$ 个阶段足够，从而总的运行时间为 $O(f \cdot |E|)$ ，因为通过无权最短路径算法一条增长路径可以以 $O(|E|)$ 时间找到。

370 图9-47所示为说明这个运行时间不好的经典例子。

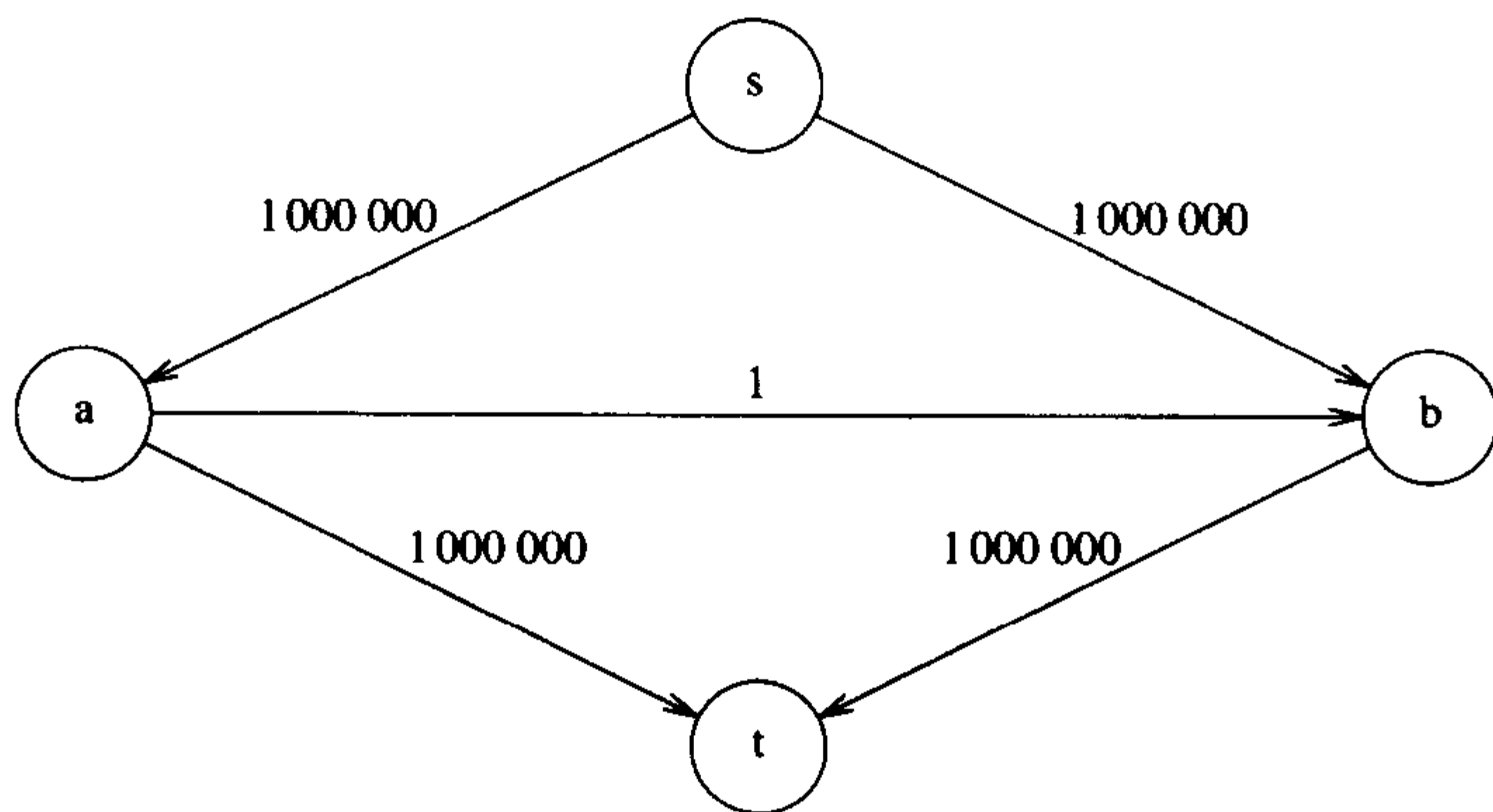


图9-47 经典的不好的增长情形

最大流可以通过沿每条边发送1 000 000并查验到2 000 000看出。随机的增长路径可以沿包含由 $a$ 和 $b$ 连接的边的路径连续增长。如果这种情况重复发生，那就需要2 000 000条增长路径，而此时仅用2条增长路径就可得出最大流。

避免这个问题的简单方法是总选择使得流增长最大的增长路径。寻找这样一条路径类似于求解一个加权最短路径问题，而对Dijkstra算法的单线修改将会完成这项工作。如果 $cap_{\max}$ 为最大边

容量，那么可以证明， $O(|E|\log \text{cap}_{\max})$ 条增长路径将足以找到最大流。在这种情况下，由于对于增长路径的每一次计算都需要 $O(|E|\log |V|)$ 时间，因此总的时间界为 $O(|E|^2 \log |V| \log \text{cap}_{\max})$ 。如果容量均为小整数，则该界可以减为 $O(|E|^2 \log |V|)$ 。

另一种选择增长路径的方法是总选取具有最少边数的路径，有理由期望，通过以这种方式选择路径不太可能使路径上出现一条小的、限制流的边。使用这种法则，可以证明需要 $O(|E||V|)$ 步增长，每一步花费 $O(|E|)$ 时间，再使用无权最短路径算法，则产生运行时间界为 $O(|E|^2|V|)$ 。

有可能对这一算法进行进一步的数据结构改进，存在几个更加复杂的算法。长期以来对界的改进降低了该问题当前熟知的界。虽然尚未见到 $O(|E||V|)$ 算法的报告，但是一些具有界 $O(|E||V|\log(|V|^2/|E|))$ 和 $O(|E||V|+|V|^{2+\epsilon})$ 的算法已经被发现（见参考文献）。还有许多在一些特殊情形下非常好的界。例如，若图除源点和汇点外所有的顶点都有一条容量为1的入边或一条容量为1的出边，则该图的最大流可以以时间 $O(|E||V|^{1/2})$ 找到。这些图出现在许多应用中。

产生这些界的分析过程是相当复杂的，并且还不清楚最坏情形的结果是如何与实际中用到的运行时间发生关系的。一个相关的甚至更困难的问题是**最小值流**（min-cost flow）问题。每条边不仅有容量，而且还有每个单位的流的值，而问题则是在所有的最大流中找出一个最小值的流来。目前对这两个问题的研究都在积极地进行。

## 9.5 最小生成树

我们将要考虑的下一个问题是在无向图中找出一棵**最小生成树**（minimum spanning tree）。这个问题对有向图也是有意义的，不过找起来更困难。大体上说来，一个无向图 $G$ 的最小生成树就是由该图的那些连接 $G$ 的所有顶点的边构成的树，且其总值最低。最小生成树存在当且仅当 $G$ 是连通的。虽然一个健壮的算法应该指出 $G$ 不连通的情况，但是我们还是假设 $G$ 是连通的，而把算法的健壮性作为练习留给读者。

在图9-48中，第二个图是第一个图的最小生成树（碰巧还是唯一的，但这并不代表一般情况）。注意，在最小生成树中边的条数为 $|V|-1$ 。最小生成树是一棵树，因为它无环；因为最小生成树包含每一个顶点，所以它是生成树；此外，它显然是包含图的所有顶点的最小的树。如果我们需要用最少的电线给一所房子安装电路，那就需要解决最小生成树问题。

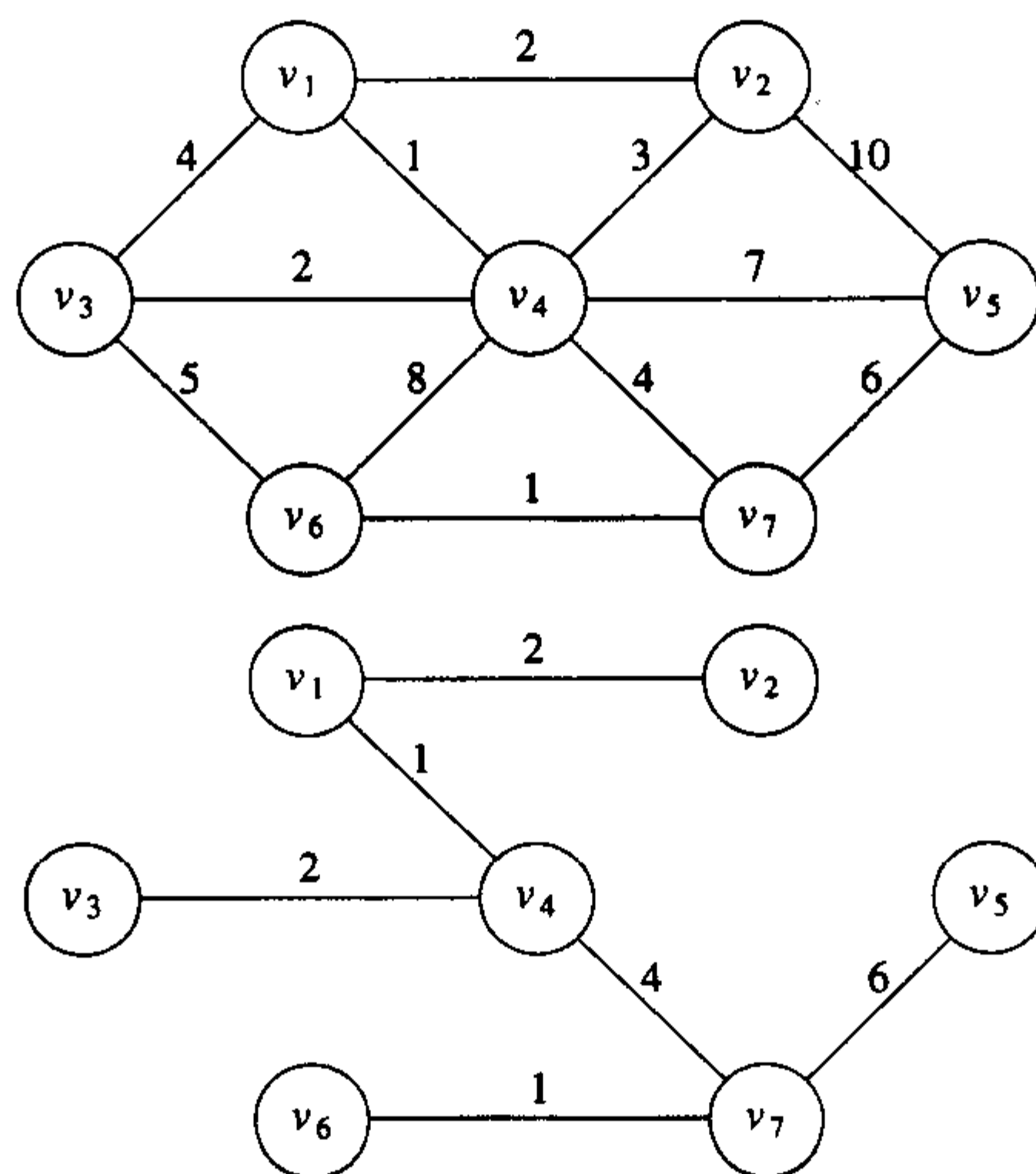


图9-48 图 $G$ 和它的最小生成树



对于任一生成树 $T$ ，如果将一条不属于 $T$ 的边 $e$ 添加进来，则产生一个回路。如果从该回路中除去任意一条边，则又恢复生成树的特性。如果边 $e$ 的值比除去的边的值小，那么新的生成树的值就比原生成树的值小。如果在建立生成树时所添加的边在所有避免成回路的边中值最小，那么最后得到的生成树的值不能再改进，因为任意一条替代的边都至少与已经存在于该生成树中的一条边具有相同的值。这说明，对于最小生成树，贪心的做法是成立的。我们介绍两种算法，它们的区别在于最小（值的）边的选取上。

### 9.5.1 Prim算法

计算最小生成树的一种方法是使其连续地一步步长成。在每一步，都要把一个结点当作根并往上加边，这样也就把相关联的顶点加到增长中的树上了。

在算法的任一时刻，我们都可以看到一个已经添加到树上的顶点集，而其余顶点尚未加到这棵树中。此时，算法在每一阶段都可以通过选择边 $(u,v)$ ，使得 $(u,v)$ 的值是所有 $u$ 在树上、但 $v$ 不在树上的边的值中的最小者，而找出一个新的顶点并把它添加到这棵树中。图9-49指出该算法如何从 $v_1$ 开始构建最小生成树。开始时， $v_1$ 在构建中的树上，它作为树的根但是没有边。每一步添加一条边和一个顶点到树上。

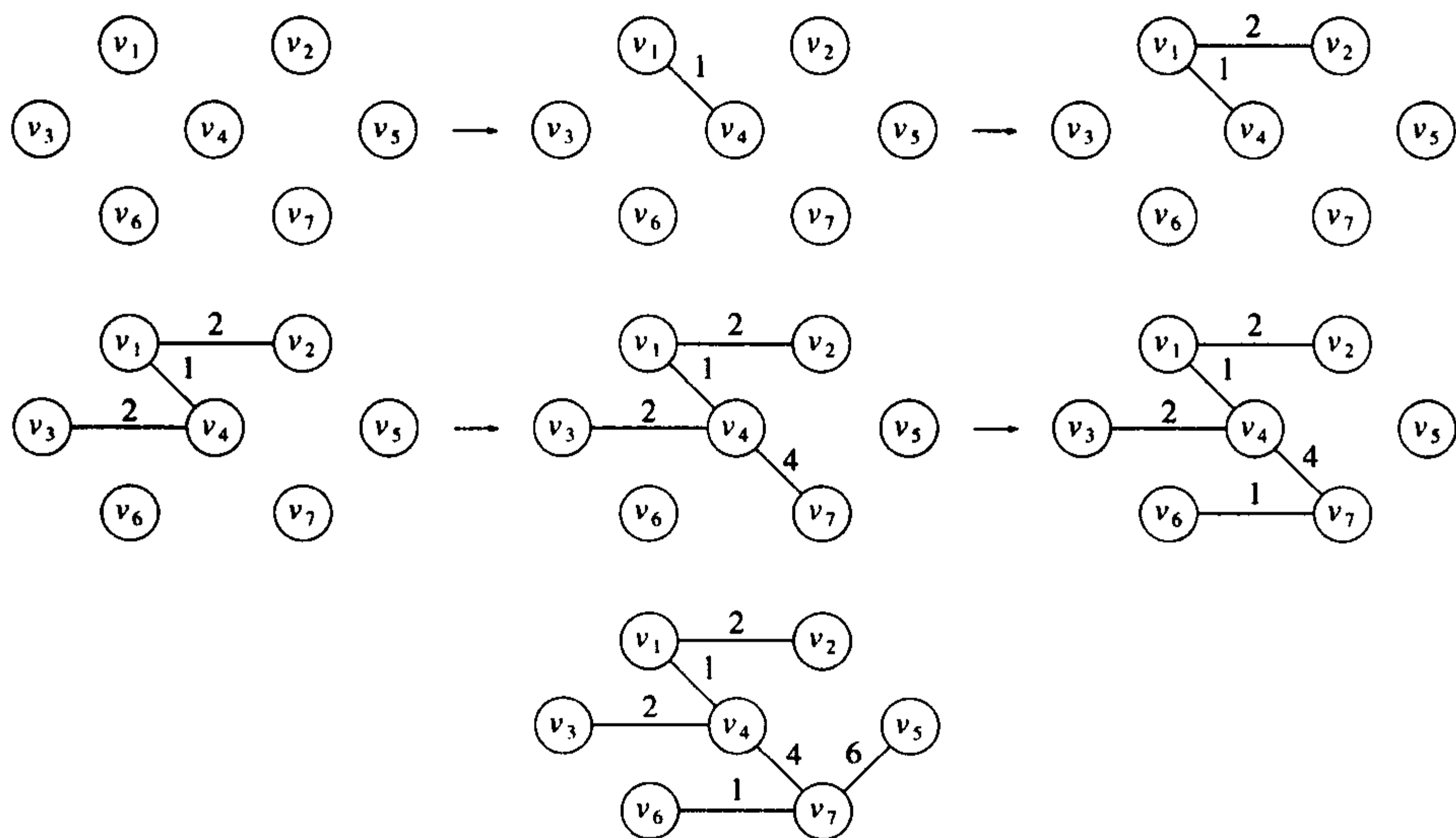


图9-49 在每一步之后的Prim算法

可以看到，Prim算法基本上和求最短路径的Dijkstra算法相同，因此和前面一样，我们对每一个顶点保留值 $d_v$ 和 $p_v$ 以及一个指标，标示该顶点是 $known$ 还是 $unknown$ 。这里， $d_v$ 是连接 $v$ 到已知顶点的最短边的权，而 $p_v$ 则是导致 $d_v$ 改变的最后的顶点。算法的其余部分完全一样，只有一点不同：由于 $d_v$ 的定义不同，因此它的更新法则也不同。事实上，更新法则比以前更简单：在每一个顶点 $v$ 被选取以后，对于每一个与 $v$ 邻接的未知的 $w$ ， $d_w = \min(d_w, c_{w,v})$ 。

表的初始状态如图9-50所示。 $v_1$ 被选取， $v_2$ 、 $v_3$ 、 $v_4$ 被更新。结果如图9-51所示。下一个顶点选取 $v_4$ ，每一个顶点都与 $v_4$ 邻接。 $v_1$ 不考虑，因为它为 $known$ 。 $v_2$ 不变，因为 $d_v = 2$ 而且从 $v_4$ 到 $v_2$ 的边的值是3；所有其他的顶点都被更新。图9-52显示得到的结果。下一个要选取的顶点是 $v_2$ ，这并不影响任何距离。然后选取 $v_3$ ，它影响到 $v_6$ 的距离，见图9-53。选取 $v_7$ 得到图9-54， $v_7$ 的选取迫使 $v_6$ 和 $v_5$ 进行调整。然后分别选取 $v_6$ 和 $v_5$ ，算法完成。

最后的表在图9-55中给出。生成树的边可以从该表中读出： $(v_2, v_1)$ ,  $(v_3, v_4)$ ,  $(v_4, v_1)$ ,  $(v_5, v_7)$ ,  $(v_6, v_7)$ ,  $(v_7, v_4)$ 。生成树总的值是16。

| $v$   | $known$ | $d_v$    | $p_v$ |
|-------|---------|----------|-------|
| $v_1$ | F       | 0        | 0     |
| $v_2$ | F       | $\infty$ | 0     |
| $v_3$ | F       | $\infty$ | 0     |
| $v_4$ | F       | $\infty$ | 0     |
| $v_5$ | F       | $\infty$ | 0     |
| $v_6$ | F       | $\infty$ | 0     |
| $v_7$ | F       | $\infty$ | 0     |

图9-50 在Prim算法中使用的表的初始状态

| $v$   | $known$ | $d_v$    | $p_v$ |
|-------|---------|----------|-------|
| $v_1$ | T       | 0        | 0     |
| $v_2$ | F       | 2        | $v_1$ |
| $v_3$ | F       | 4        | $v_1$ |
| $v_4$ | F       | 1        | $v_1$ |
| $v_5$ | F       | $\infty$ | 0     |
| $v_6$ | F       | $\infty$ | 0     |
| $v_7$ | F       | $\infty$ | 0     |

图9-51 在 $v_1$ 声明为已知 ( $known$ ) 后的表

| $v$   | $known$ | $d_v$ | $p_v$ |
|-------|---------|-------|-------|
| $v_1$ | T       | 0     | 0     |
| $v_2$ | F       | 2     | $v_1$ |
| $v_3$ | F       | 2     | $v_4$ |
| $v_4$ | T       | 1     | $v_1$ |
| $v_5$ | F       | 7     | $v_4$ |
| $v_6$ | F       | 8     | $v_4$ |
| $v_7$ | F       | 4     | $v_4$ |

图9-52 在 $v_4$ 声明为已知后的表

| $v$   | $known$ | $d_v$ | $p_v$ |
|-------|---------|-------|-------|
| $v_1$ | T       | 0     | 0     |
| $v_2$ | T       | 2     | $v_1$ |
| $v_3$ | T       | 2     | $v_4$ |
| $v_4$ | T       | 1     | $v_1$ |
| $v_5$ | F       | 7     | $v_4$ |
| $v_6$ | F       | 5     | $v_3$ |
| $v_7$ | F       | 4     | $v_4$ |

图9-53 在 $v_2$ 和 $v_3$ 先后声明为已知后的表

| $v$   | $known$ | $d_v$ | $p_v$ |
|-------|---------|-------|-------|
| $v_1$ | T       | 0     | 0     |
| $v_2$ | T       | 2     | $v_1$ |
| $v_3$ | T       | 2     | $v_4$ |
| $v_4$ | T       | 1     | $v_1$ |
| $v_5$ | F       | 6     | $v_7$ |
| $v_6$ | F       | 1     | $v_7$ |
| $v_7$ | T       | 4     | $v_4$ |

图9-54 在 $v_7$ 声明为已知后的表

| $v$   | $known$ | $d_v$ | $p_v$ |
|-------|---------|-------|-------|
| $v_1$ | T       | 0     | 0     |
| $v_2$ | T       | 2     | $v_1$ |
| $v_3$ | T       | 2     | $v_4$ |
| $v_4$ | T       | 1     | $v_1$ |
| $v_5$ | T       | 6     | $v_7$ |
| $v_6$ | T       | 1     | $v_7$ |
| $v_7$ | T       | 4     | $v_4$ |

图9-55 在 $v_6$ 和 $v_5$ 选取后的表 (Prim算法终止)

该算法整个的实现实际上和Dijkstra算法的实现是一样的, 对于Dijkstra算法所做的分析都可以用到这里。不过要注意, Prim算法是在无向图上运行的, 因此当编写代码的时候要记住把每一条边都放到两个邻接表中。不用堆时, 运行时间为 $O(V^2)$ , 它对于稠密的图来说是最优的。使用二叉堆时, 运行时间是 $O(E \log V)$ , 对于稀疏的图这是一个好的界。

### 9.5.2 Kruskal算法

第二种贪心策略是连续地按照最小的权选择边, 并且当所选的边不产生回路时就把它作为取定的边。该算法对于前面例子中的图的实现过程如图9-56所示。

| 边            | 权 | 动作 |
|--------------|---|----|
| $(v_1, v_4)$ | 1 | 接受 |
| $(v_6, v_7)$ | 1 | 接受 |
| $(v_1, v_2)$ | 2 | 接受 |
| $(v_3, v_4)$ | 2 | 接受 |
| $(v_2, v_4)$ | 3 | 拒绝 |
| $(v_1, v_3)$ | 4 | 拒绝 |
| $(v_4, v_7)$ | 4 | 接受 |
| $(v_3, v_6)$ | 5 | 拒绝 |
| $(v_5, v_7)$ | 6 | 接受 |

图9-56 Kruskal算法施于图G的过程

形式上, **Kruskal**算法是在处理一个森林——树的集合。开始的时候, 存在 $|V|$ 棵单结点树, 而添加一边则将两棵树合并成一棵树。当算法终止的时候, 就只有一棵树了, 这棵树就是最小生成树。图9-57显示了边被添加到森林中的顺序。

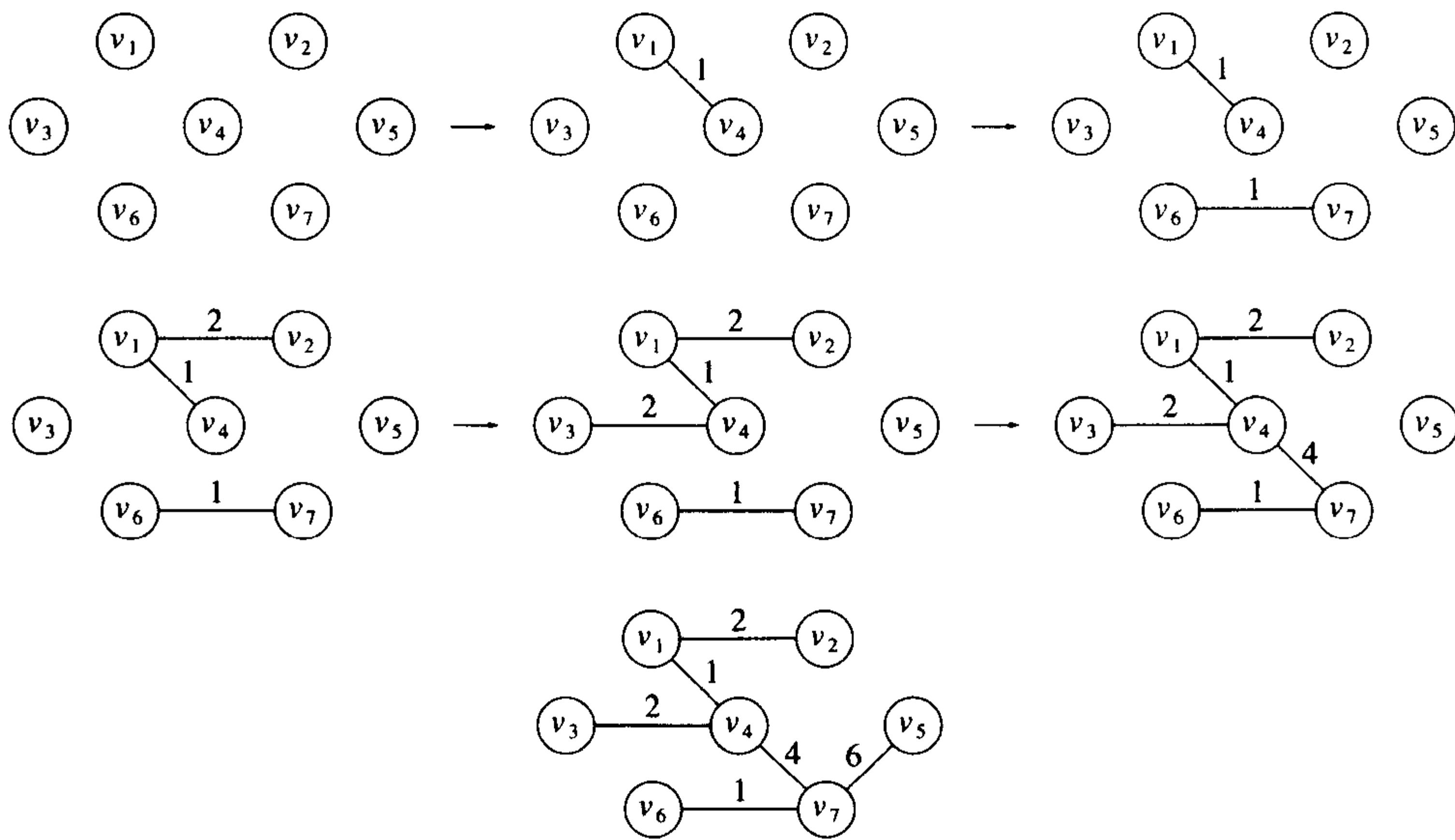


图9-57 在每一步之后的Kruskal算法

当添加到森林中的边足够多时, 算法终止。实际上, 算法就是要决定边 $(u, v)$ 应该添加还是舍弃。第8章中的Union/Find算法在这里也适用。

我们利用的一个恒定的事实是, 在算法实施的任一时刻, 两个顶点属于同一个集合当且仅当它们在当前的生成森林 (spanning forest) 中连通。因此, 每个顶点最初是在它自己的集合中。如果 $u$ 和 $v$ 在同一个集合中, 那么连接它们的边就要放弃, 因为它们已经连通了, 因此再添加边 $(u, v)$ 就会形成一个回路。如果这两个顶点不在同一个集合中, 则将该边加入, 并对包含顶点 $u$ 和 $v$ 的这两个集合实施一次union。容易看到, 这样将保持集合不变性, 因为一旦边 $(u, v)$ 添加到生成森林中, 若 $w$ 连通到 $u$ 而 $x$ 连通到 $v$ , 则 $x$ 和 $w$ 必然是连通的, 因此属于相同的集合。

虽然将边排序便于选取, 但是, 用线性时间建立一个堆则是更好的想法。此时, deleteMin将使得边依序得到测试。一般情况下, 在算法终止前只有一小部分边需要测试, 尽管必须测试所

有边的情况也是有可能的。例如，假设还有一个顶点 $v_8$ 以及值为100的边 $(v_5, v_8)$ ，那么所有的边就都要考察到。图9-58中的方法kruskal可以找出一棵最小生成树。

```
void Graph::kruskal( )
{
    int edgesAccepted = 0;
    DisjSet ds( NUM_VERTICES );
    PriorityQueue<Edge> pq( getEdges( ) );
    Edge e;
    Vertex u, v;

    while( edgesAccepted < NUM_VERTICES - 1 )
    {
        pq.deleteMin( e );      // Edge e = (u, v)
        SetType uset = ds.find( u );
        SetType vset = ds.find( v );
        if( uset != vset )
        {
            // Accept the edge
            edgesAccepted++;
            ds.unionSets( uset, vset );
        }
    }
}
```

图9-58 Kruskal算法的伪代码

该算法的最坏情形运行时间为 $O(|E|\log|E|)$ ，它受堆操作控制。注意，由于 $|E|=O(|V|^2)$ ，因此这个运行时间实际上是 $O(|E|\log|V|)$ 。在实践中，该算法要比这个时间界指示的时间快得多。

## 9.6 深度优先搜索的应用

深度优先搜索（depth-first search）是对前序遍历的推广。我们从某个顶点 $v$ 开始处理 $v$ ，然后递归地遍历所有与 $v$ 邻接的顶点。如果这个过程是对一棵树进行，那么，由于 $|E|=\Theta(|V|)$ ，因此该树的所有顶点在总时间 $O(|E|)$ 内都将被系统地访问到。如果对任意的图执行该过程，那么为了避免回路，我们需要小心仔细。为此，当访问一个顶点 $v$ 的时候，由于已经到了该点处，因此可以标记该点是访问过的，并且对于尚未被标记的所有邻接顶点递归地调用深度优先搜索。我们假设，对于无向图，每条边 $(v, w)$ 在邻接表中出现两次：一次是 $(v, w)$ ，另一次是 $(w, v)$ 。图9-59中的过程执行一次深度优先搜索（此外绝对什么也不做），从而是一个通用风格的模板。

```
void Graph::dfs( Vertex v )
{
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}
```

图9-59 深度优先搜索模板（伪代码）

对每一个顶点，字段visited初始化成false。通过只对那些尚未被访问的结点递归调用该过程，我们保证不会陷入无限的循环。如果图是无向的且不连通，或是有向的但非强连通，这种方法可能会访问不到某些结点。此时，我们搜索一个未被标记的结点，然后应用深度优先遍历，



并继续这个过程直到不存在未标记的结点为止<sup>1</sup>。因为该方法保证每一条边只访问一次，所以只要使用邻接表，执行遍历的总时间就是 $O(|E|+|V|)$ 。

### 9.6.1 无向图

无向图是连通的，当且仅当从任一结点开始的深度优先搜索访问到每一个结点。因为这项测试应用起来非常容易，所以假设我们处理的图都是连通的。如果它们不连通，那么可以找出所有的连通分支并将算法依次应用于每个分支。

作为深度优先搜索的一个例子，设从图9-60中的A点开始。此时，标记A为访问过的并递归调用dfs(B)。dfs(B)标记B为访问过的并递归调用dfs(C)。dfs(C)标记C为访问过的并递归调用dfs(D)。dfs(D)遇到A和B，但是这两个结点都已经被标记过了，因此没有递归调用可以进行。dfs(D)还看到C是邻接的顶点，但C也标记过了，因此在这里也没有递归调用进行，于是dfs(D)返回到dfs(C)。dfs(C)看到B是邻接点，忽略它，并发现以前没看见的顶点E也是邻接点，因此调用dfs(E)。dfs(E)标记E，忽略A和C，并返回到dfs(C)。dfs(C)返回到dfs(B)。dfs(B)忽略A和D并返回。dfs(A)忽略D和E且返回（我们实际上已经接触每条边两次，一次是作为边 $(v, w)$ ，再一次是作为边 $(w, v)$ ，但这实际上是每个邻接表项接触一次）。

我们以图形来描述深度优先生成树（depth-first spanning tree）的步骤。该树的根是A，是第一个被访问到的顶点。图中的每一条边 $(v, w)$ 都出现在树上。如果处理 $(v, w)$ 时发现w是未被标记的，或者处理 $(w, v)$ 时发现v是未被标记的，那么我们就用树的一条边表示它。如果处理 $(v, w)$ 时发现w已被标记，并且处理 $(w, v)$ 时发现v也已被标记，那么我们就画一条虚线，并称之为后向边（back edge），表示这条“边”实际上不是树的一部分。图9-60的深度优先搜索在图9-61中给出。

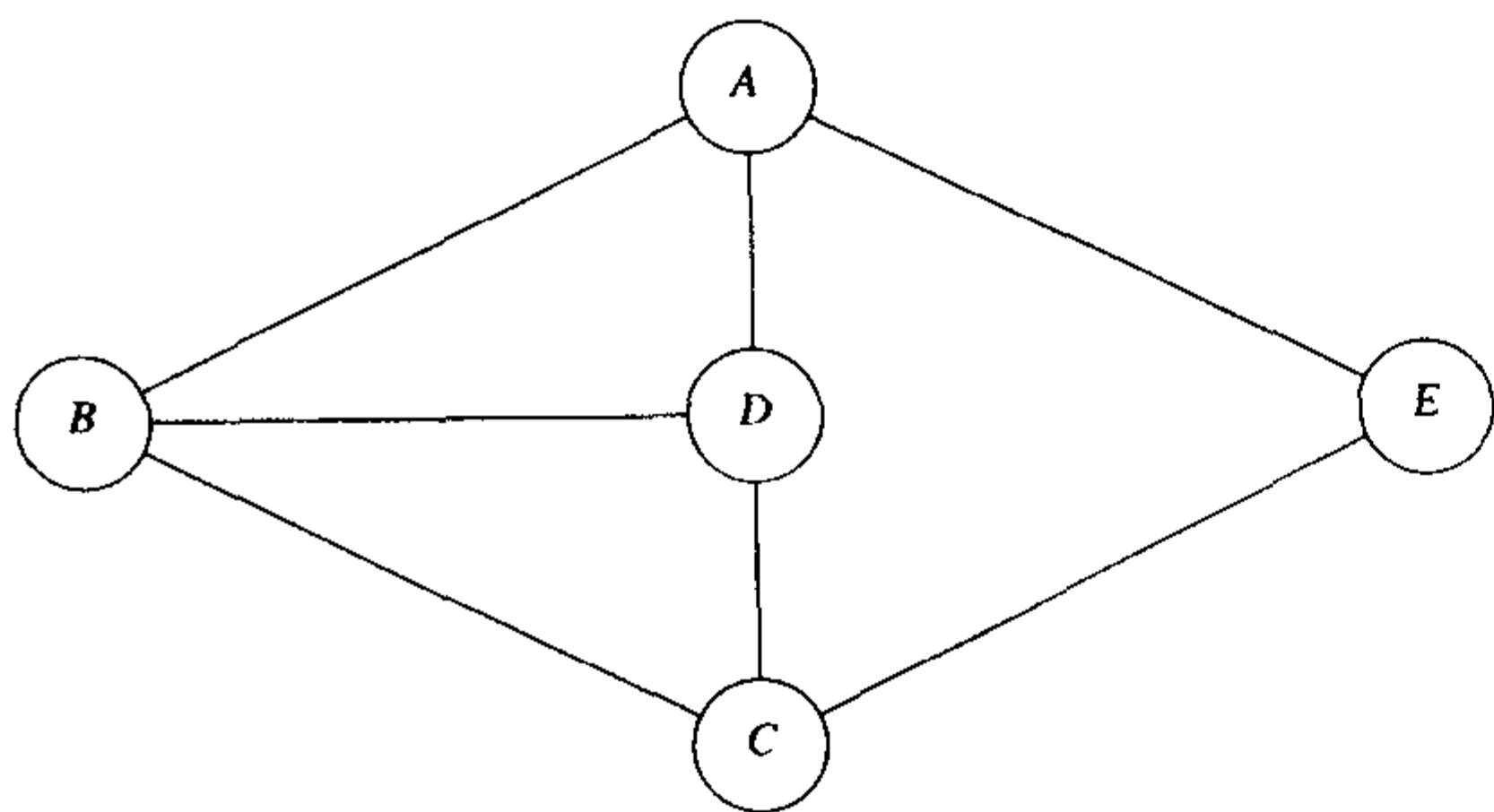


图9-60 一个无向图

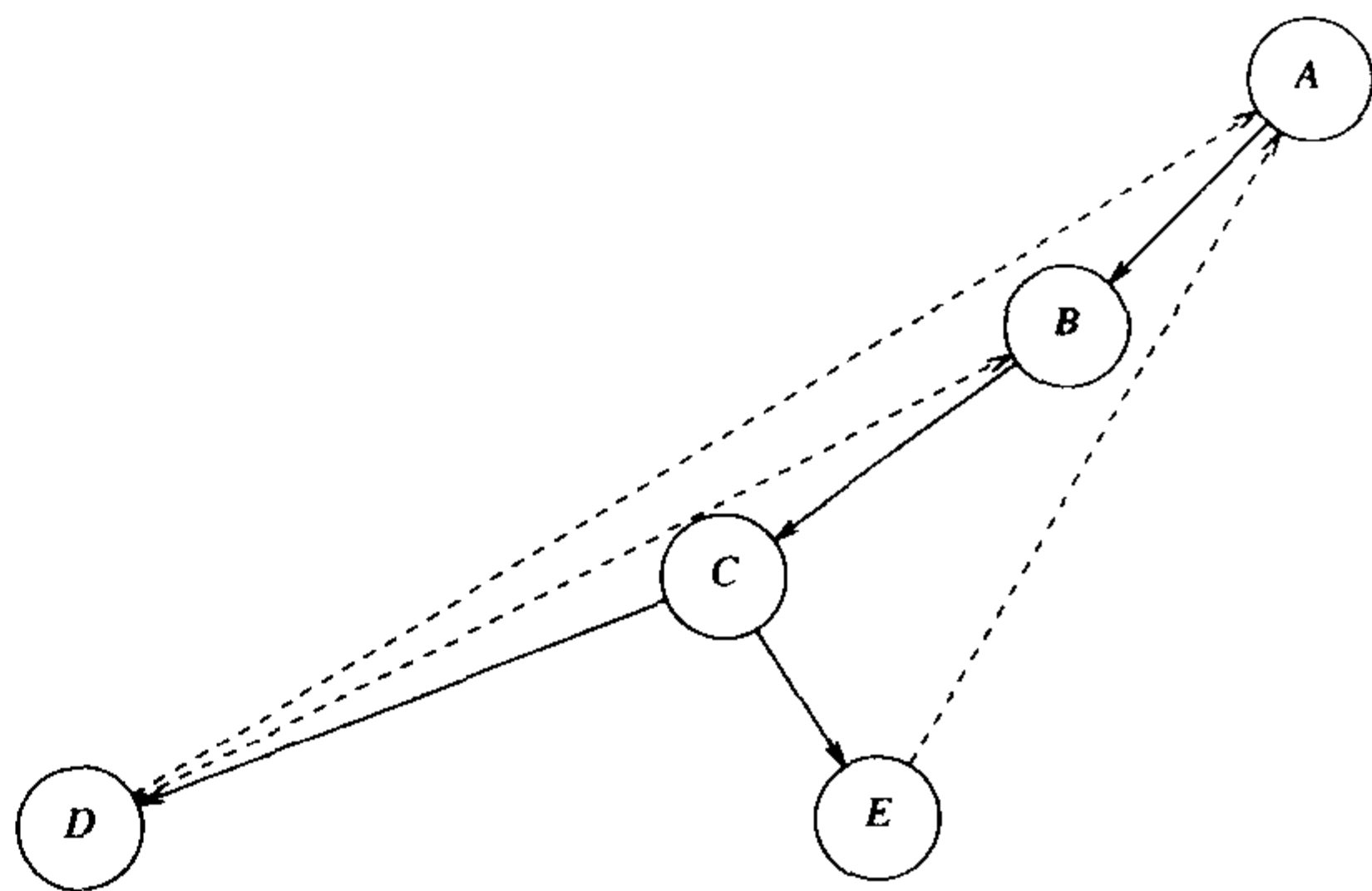


图9-61 图9-60的深度优先搜索

树将模拟我们执行的遍历。只使用树的边对该树的前序编号（preorder numbering）告诉我们这些顶点被标记的顺序。如果图不是连通的，那么处理所有的结点（和边）则需要多次调用dfs，每次都生成一棵树，整个集合就是深度优先生成森林（depth-first spanning forest）。

### 9.6.2 双连通性

一个连通的无向图中的任一顶点删除之后，剩下的图仍然连通，那么这样的无向连通图就称

1. 其实现的一种高效方法是从 $v_1$ 开始深度优先搜索。如果我们需要重新开始深度优先搜索，则对于一个未标记的顶点考查序列 $v_k, v_{k+1}, \dots$ ，其中 $v_{k-1}$ 是最后一次深度优先搜索开始时的顶点。这保证整个算法只花费 $O(|V|)$ 时间查找那些使新的深度优先搜索树开始的顶点。

为是双连通的 (biconnected)。上例中的图是双连通的。如果例中的结点是计算机，边是链路，那么，若任一台计算机出故障而不能运行，网络邮件并不受影响，当然，与这台出故障的计算机有关的邮件除外。类似地，如果一个公交运输系统是双连通的，那么，若某个站点被破坏，则用户总可以选择另外的旅行路径。

如果图不是双连通的，那么，将其删除后图不再连通的那些顶点叫作割点 (articulation point)。这些结点在许多应用中是很重要的。图9-62不是双连通的：顶点C和D是割点。删除顶点D使图G不连通，而删除顶点D则使E和F从图G的其余部分分离。

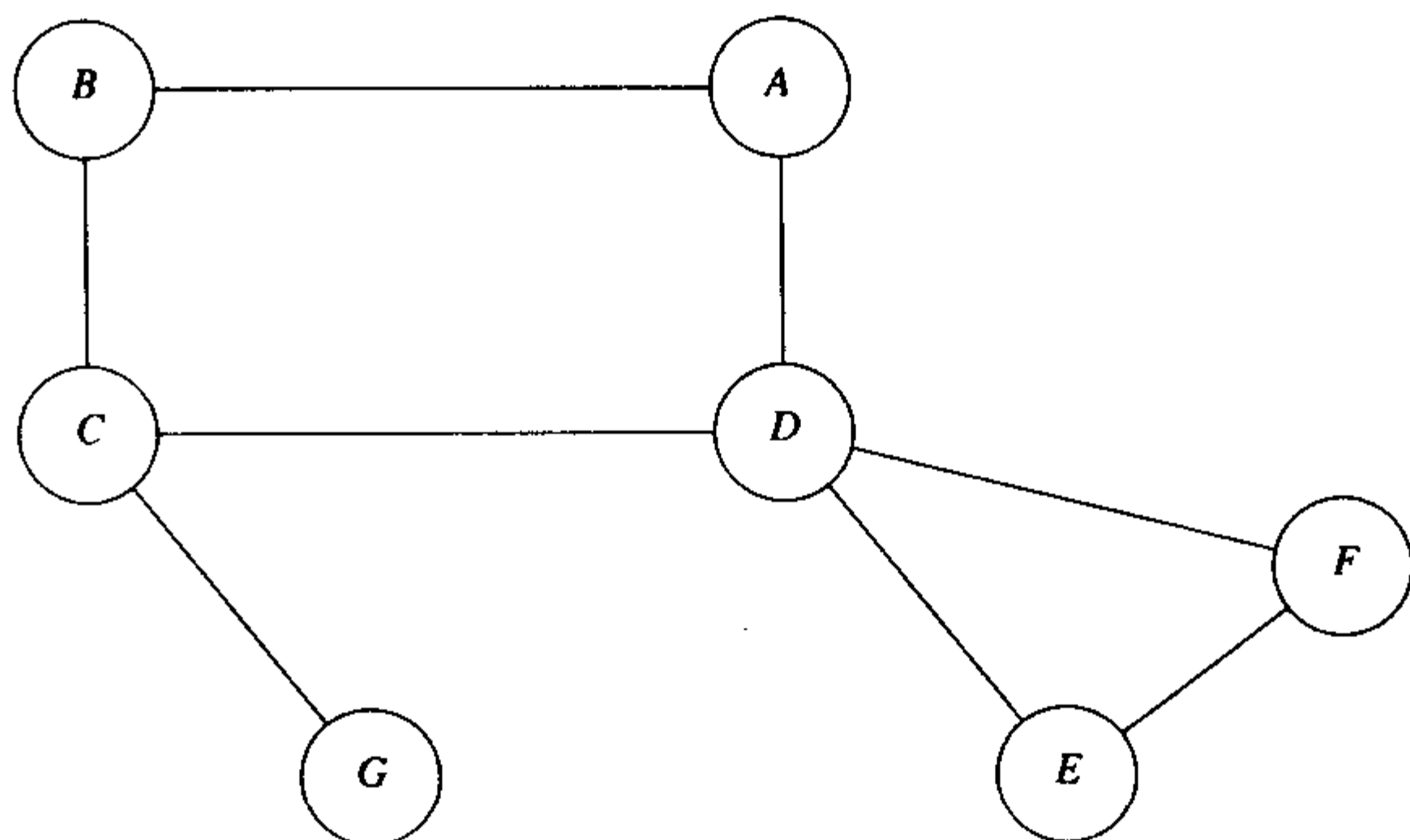


图9-62 具有割点C和D的图

深度优先搜索提供了一种找出连通图中所有割点的线性时间算法。首先，从图中任一顶点开始，执行深度优先搜索并在顶点被访问时给它们编号。对于每一个顶点 $v$ ，我们称其前序编号为 $\text{Num}(v)$ 。然后，对于深度优先搜索生成树上的每一个顶点 $v$ ，计算编号最低的顶点，我们称之为 $\text{Low}(v)$ ，该点可从 $v$ 开始通过树的零条或多条边，且可能还有一条后向边而（以该序）达到。图9-63中的深度优先搜索树首先指出前序编号，然后指出在上述法则下可达到的最低编号顶点。

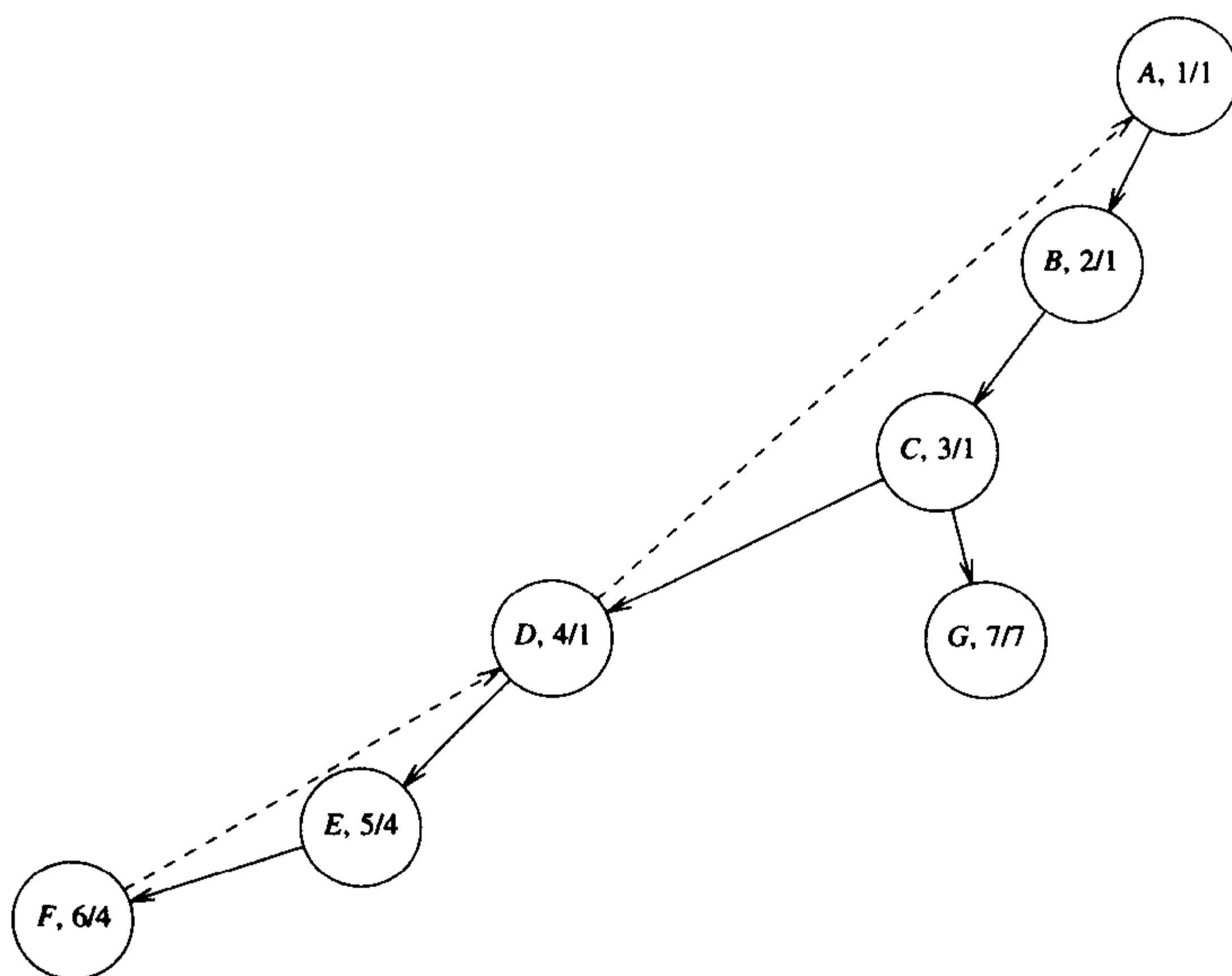


图9-63 图9-62的深度优先树，顶点标有Num和Low

从A、B和C开始的可达到的最低编号顶点为1(A)，因为它们都能够通过树的边到D，然后再

由一条后向边回到 $A$ 。我们可以通过对该深度优先生成树执行一次后序遍历有效地算出 $Low$ 。根据 $Low$ 的定义可知,  $Low(v)$ 是

- (1)  $Num(v)$ 。
- (2) 所有后向边 $(v, w)$ 中的最低 $Num(w)$ 。
- (3) 树的所有边 $(v, w)$ 中的最低 $Low(w)$ 。

中的最小者。

第一种方法是不选取边, 第二种方法是不选取树的边而是选取一条后向边, 第三种方法则是选择树的某些边以及可能还有一条后向边。第三种方法可用一个递归调用简明地描述。由于我们需要对 $v$ 的所有儿子计算出 $Low$ 值后才能计算 $Low(v)$ , 因此这是一个后序遍历。对于任一条边 $(v, w)$ , 只要检查 $Num(v)$ 和 $Num(w)$ 就可以知道它是一条树的边还是一条后向边。因此,  $Low(v)$ 容易计算: 我们只需扫描 $v$ 的邻接表, 应用适当的法则, 并记住最小者。所有的计算花费 $O(|E|+|V|)$ 时间。

剩下要做的就是利用这些信息找出所有的割点。根是割点当且仅当它有多个儿子, 因为如果它有两个儿子, 那么删除根则使得结点不连通而分布在不同的子树上; 如果根只有一个儿子, 那么除去该根只不过是断离该根。对于任何其他顶点 $v$ , 当且仅当它有某个儿子 $w$ 使得 $Low(w) \geq Num(v)$ 才是割点。注意, 这个条件在根处总是满足的; 因此, 需要进行特别的测试。

382

当我们考察算法确定的割点, 即 $C$ 和 $D$ 时, 证明的“if”(当)部分是明显的。 $D$ 有一个儿子 $E$ , 且 $Low(E) \geq Num(D)$ , 二者都是4。因此, 对 $E$ 来说只有一种方法到达 $D$ 上面的任何一点, 那就是要通过 $D$ 。类似地,  $C$ 也是一个割点, 因为 $Low(G) \geq Num(C)$ 。为了证明该算法正确, 我们必须证明论断的“only if”(仅当)部分成立(即它找到所有的割点)。我们把它留作练习。作为第二个例子, 我们指出(见图9-64)在相同的图上应用该算法从顶点 $C$ 开始深度优先搜索的结果。

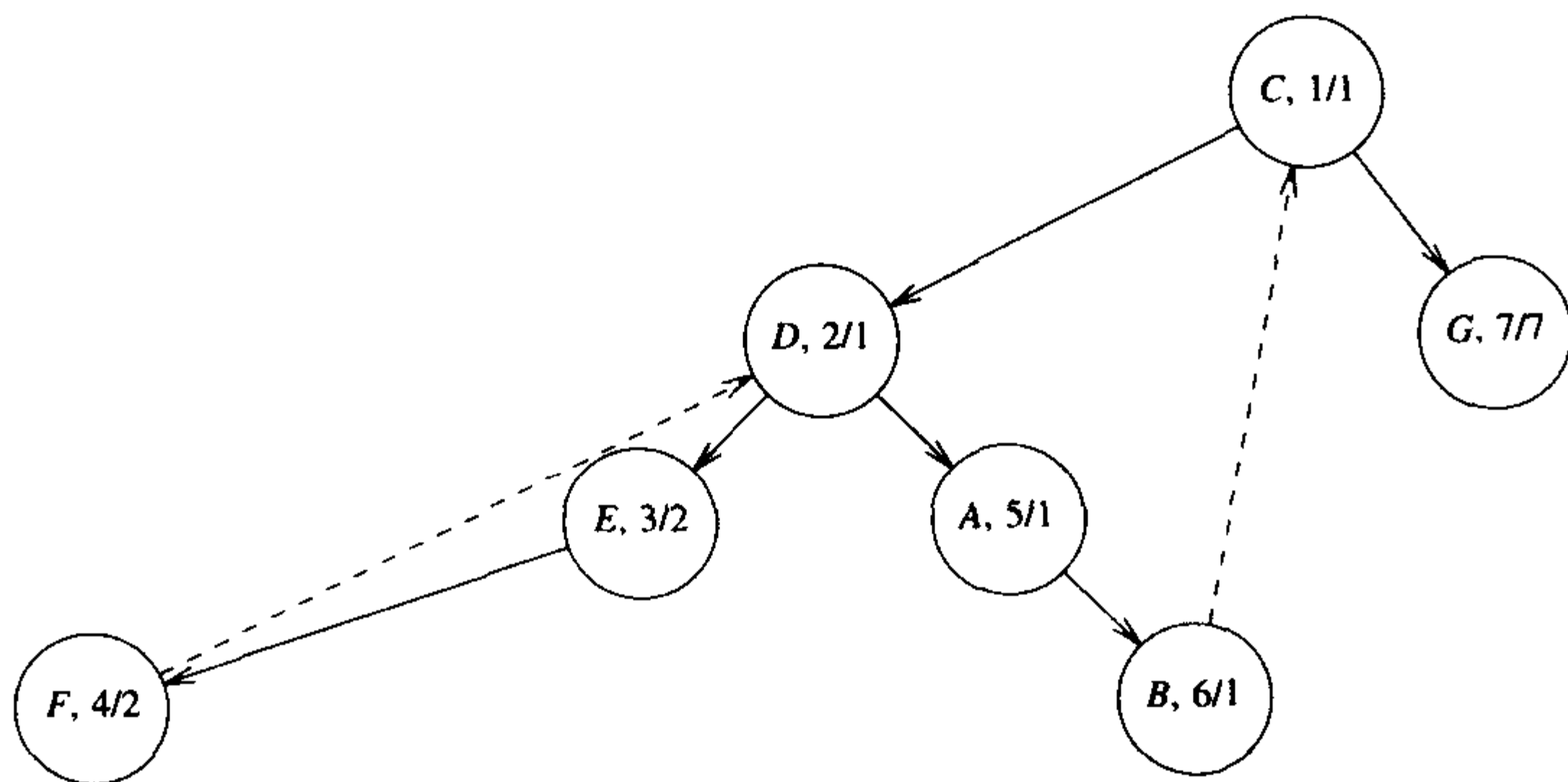


图9-64 从 $C$ 开始深度优先搜索所得到的深度优先树

最后, 我们给出伪代码实现该算法。设 $Vertex$ 包含数据字段 $visited$ (初始化为 $false$ )、 $num$ 、 $low$ 和 $parent$ 。还有一个( $Graph$ )类变量叫作 $counter$ , 用来给前序遍历编号 $num$ 赋值, 将 $counter$ 初始化为1。这里省略对根的容易实现的测试。

正如前面已经提到的, 该算法可以通过执行一次前序遍历计算 $Num$ , 而后执行一次后序遍历计算 $Low$ 来实现。第三趟遍历可以用来检验哪些顶点满足割点的标准。然而, 执行三趟遍历是一种浪费。第一趟如图9-65所示。

```

/**
 * Assign num and compute parents.
 */
void Graph::assignNum( Vertex v )
{
    v.num = counter++;
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
        {
            w.parent = v;
            assignNum( w );
        }
}

```

图9-65 对顶点的Num赋值的例程（伪代码）

第二趟和第三趟遍历都是后序遍历，可以通过图9-66中的代码来实现。第8行处理一个特殊的情况。如果 $w$ 邻接到 $v$ ，那么递归调用 $w$ 将发现 $v$ 邻接到 $w$ 。这不是一条后向边，而只是一条已经考虑过且需要忽略的边。在其他情况下，正如算法指定的那样，该过程计算出各low和num项的最小值。

```

/**
 * Assign low; also check for articulation points.
 */
void Graph::assignLow( Vertex v )
{
    v.low = v.num; // Rule 1
    for each Vertex w adjacent to v
    {
        if( w.num > v.num ) // Forward edge
        {
            assignLow( w );
            if( w.low >= v.num )
                cout << v << " is an articulation point" << endl;
            v.low = min( v.low, w.low ); // Rule 3
        }
        else
            if( v.parent != w ) // Back edge
                v.low = min( v.low, w.num ); // Rule 2
    }
}

```

图9-66 计算Low并检验其是否为割点的伪代码（忽略对根的检验）

不存在一个遍历一定是前序遍历或后序遍历的法则。在递归调用前和递归调用后都有可能进行处理。图9-67中的过程以一种直接的方式将两个例程assignNum和assignLow结合起来，得到过程findArt。

383  
?  
384

### 9.6.3 欧拉回路

考虑图9-68中的三个图。一个流行的游戏是用钢笔重画这些图，每条线恰好画一次。在画图的时候钢笔不要从纸上离开，一气呵成。作为一个附加的问题，要在结束画图时，使钢笔回到开始画图时的起点上。该游戏有一个非常简单的解法。如果你想尝试求解该问题，那么现在就可以试一试。

第一个图仅当起点在左下角或右下角时可以画出，而且不可能结束在起点处。第二个图容易

385



画出，它的终止点和起点相同，但是第三个图在游戏的限制条件下根本画不出来。

```
void Graph::findArt( Vertex v )
{
    v.visited = true;
    v.low = v.num = counter++; // Rule 1
    for each Vertex w adjacent to v
    {
        if( !w.visited ) // Forward edge
        {
            w.parent = v;
            findArt( w );
            if( w.low >= v.num )
                cout << v << " is an articulation point" << endl;
            v.low = min( v.low, w.low ); // Rule 3
        }
        else
            if( v.parent != w ) // Back edge
                v.low = min( v.low, w.num ); // Rule 2
    }
}
```

图9-67 在一次深度优先搜索（忽略对根的检测）中对割点的检测（伪代码）

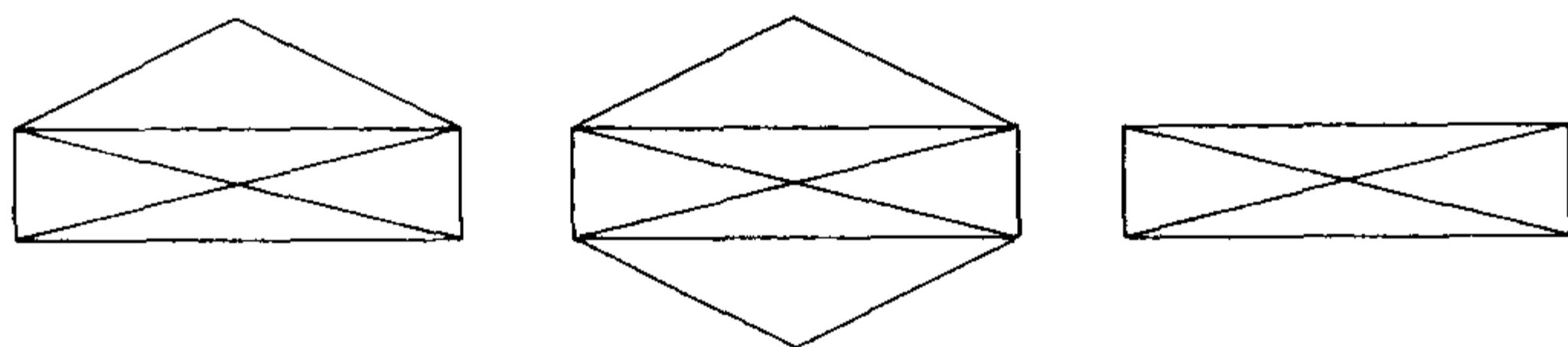


图9-68 三个图

可以通过给每个交点指定一个顶点而把这个问题转化成图论问题。此时，图的边可以以自然的方式规定，如图9-69所示。

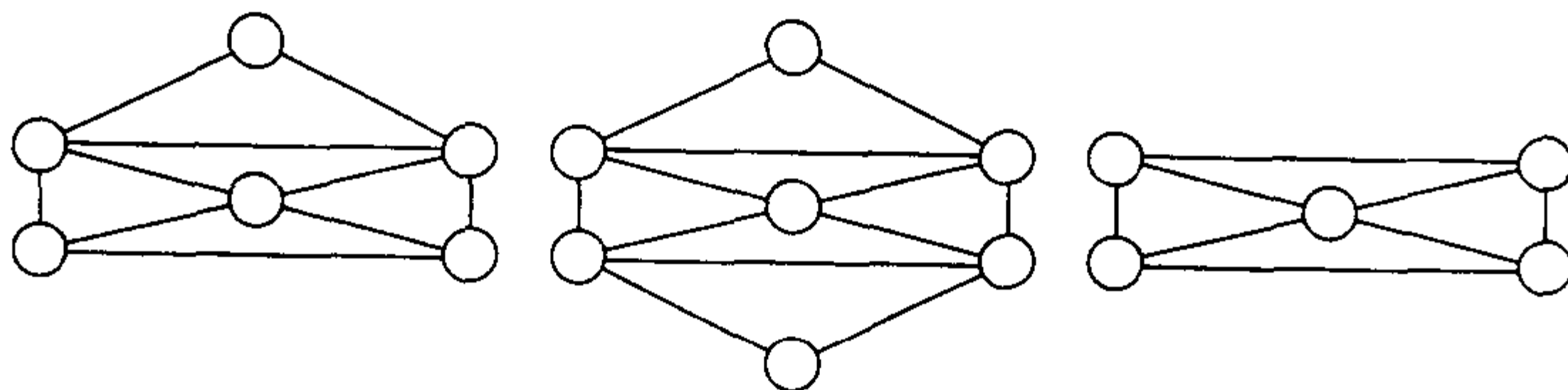


图9-69 将游戏转化成图

将问题转化之后，必须在图中找出一条路径，使得该路径对图的每条边恰好访问一次。如果我们要解决“附加的问题”，那么就必须要找到一个回路，该回路恰好经过每条边一次。这个图论问题于1736年由欧拉解出，它标志着图论的诞生。根据特定问题的叙述不同，这种问题通常叫作**欧拉路径**（有时称**欧拉环游**（Euler tour））或**欧拉回路**（Euler circuit）。虽然欧拉环游和欧拉回路问题稍有不同，但是却有相同的基本解。因此，在这一节我们将考虑欧拉回路问题。

能够做的第一个观察是，其终点必须终止在起点上的欧拉回路只有当图是连通的并且每个顶点的度（即边的条数）是偶数时才可能存在。这是因为，在欧拉回路中，一个顶点有边进入，则必然有边离开。如果任一顶点 $v$ 的度为奇数，那么最终会达到这样一种地步，即只有一条进入 $v$ 的边尚未访问到，若沿该边进入 $v$ 点，那么只能停在顶点 $v$ ，不可能再出来。如果恰好有两个顶点的度是奇数，那么当从一个奇数度的顶点出发最后终止在另一个奇数度的顶点时，仍然有可能得

到一个欧拉环游。这里，欧拉环游是必须访问图的每一边但最后不一定必须回到起点的路径。如果奇数度的顶点多于两个，那么欧拉环游也是不可能存在的。

前面的观察提供了欧拉回路存在的一个必要条件。不过，它并未告诉我们满足该性质的所有的连通图必然有一个欧拉回路，也没有指导我们如何找出欧拉回路。事实上，这个必要条件也是充分的。也就是说，所有顶点的度均为偶数的任何连通图必然有欧拉回路。不仅如此，我们还可以以线性时间找出这样一条回路。

由于可以用线性时间检测这个充分必要条件，因此可以假设我们知道存在一条欧拉回路。此时，基本算法就是执行一次深度优先搜索。有大量“明显的”解决方案但是却都行不通，我们在练习中罗列了一些。

主要问题在于，我们可能只访问了图的一部分而提前返回到起点。如果从起点出发的所有边均已用完，那么图的某些部分就会遍历不到。最容易的补救方法是找出有尚未访问的边的路径上的第一个顶点，并执行另外一次深度优先搜索。这将给出另外一个回路，把它拼接到原来的回路上。继续该过程，直到所有的边都被遍历到为止。

386

作为一个例子，考虑图9-70。容易看出，这个图有一个欧拉回路。设从顶点5开始，我们遍历5、4、10、5，此时已无路可走，图的大部分都还未遍历到。此时的情形如图9-71所示。

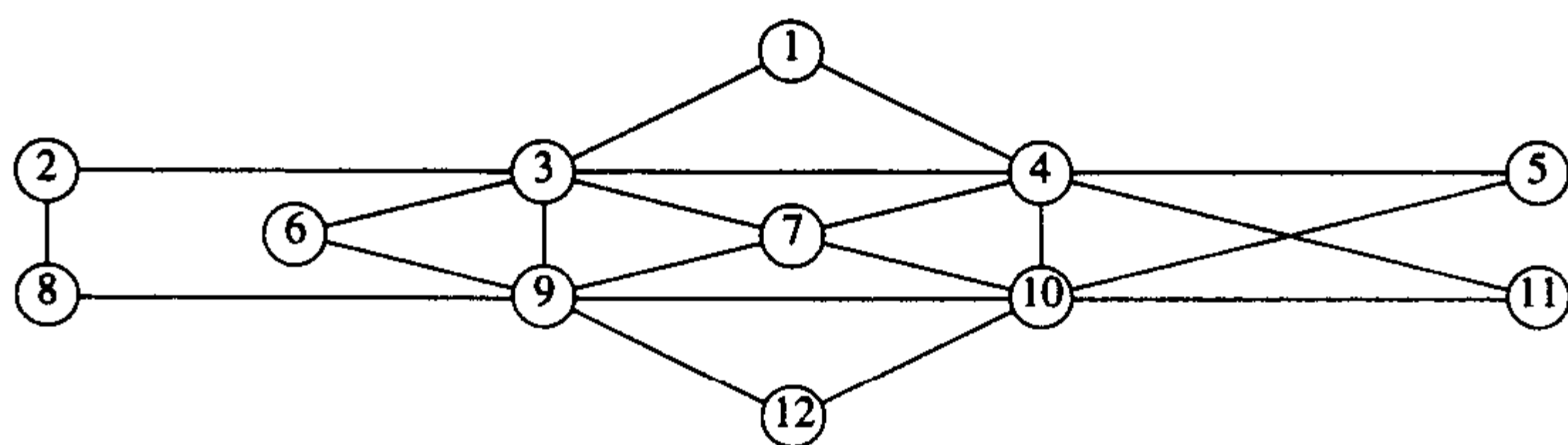


图9-70 欧拉回路问题的图

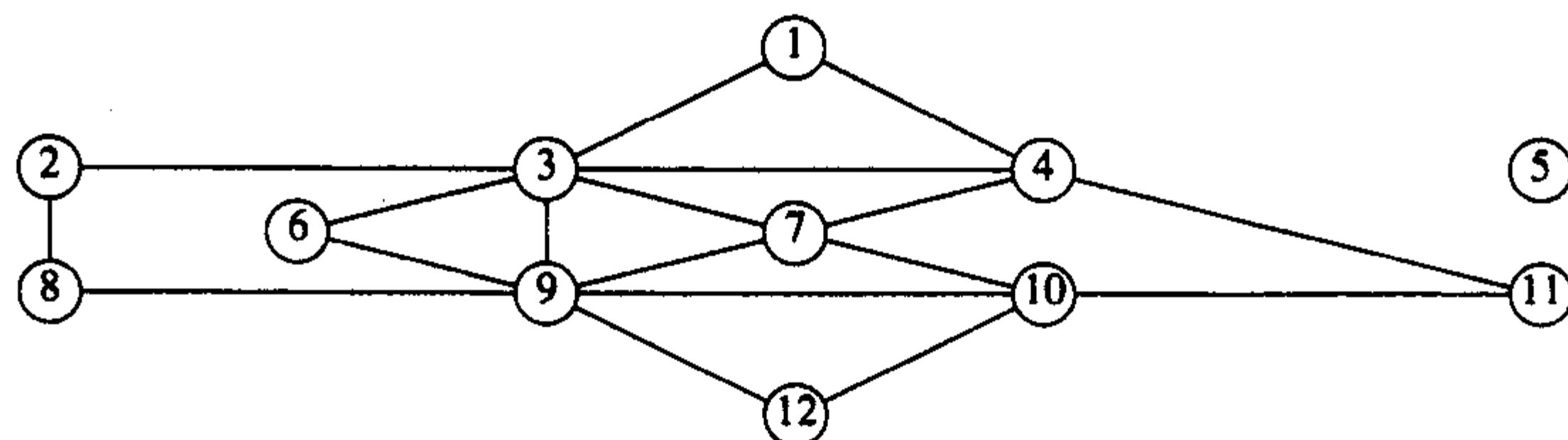


图9-71 遍历5、4、10、5后的图

此时，从顶点4继续进行，仍然还有没用到的边。结果，又得到路径4、1、3、7、4、11、10、7、9、3、4。如果把这条路径拼接到前面的路径5、4、10、5上，那么就得到一条新的路径5、4、1、3、7、4、11、10、7、9、3、4、10、5。

此后，剩下的图表示在图9-72中。注意，在这个图中，所有顶点的度必然都是偶数，因此，我们保证能够找到一个回路再拼接上。剩下的图可能不是连通的，但这并不重要。路径上存有未被访问的边的下一个顶点是3。此时可能的回路是3、2、8、9、6、3。当拼接进来之后，得到路径5、4、1、3、2、8、9、6、3、7、4、11、10、7、9、3、4、10、5。

剩下的图表示在图9-73中。在该路径上，带有未遍历的边的下一个顶点是9，算法找到回路9、12、10、9。当把它拼接到当前路径中时，得到回路5、4、1、3、2、8、9、12、10、9、6、3、7、4、11、10、7、9、3、4、10、5。当所有的边都被遍历时，算法终止，我们得到一个欧拉回路。

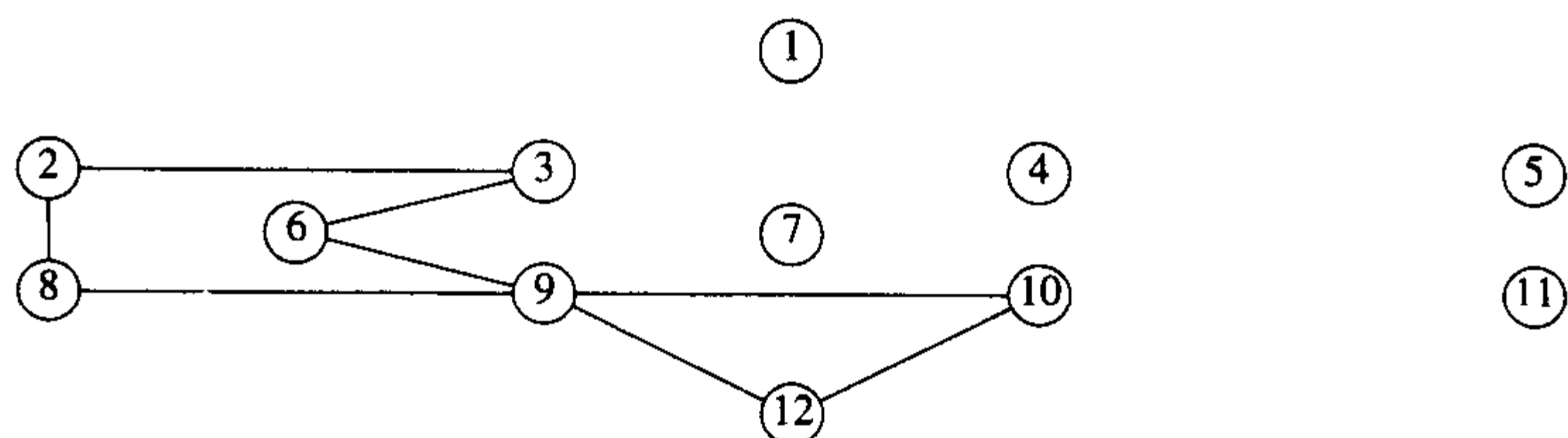


图9-72 遍历路径5、4、1、3、7、4、11、10、7、9、3、4、10、5之后的图

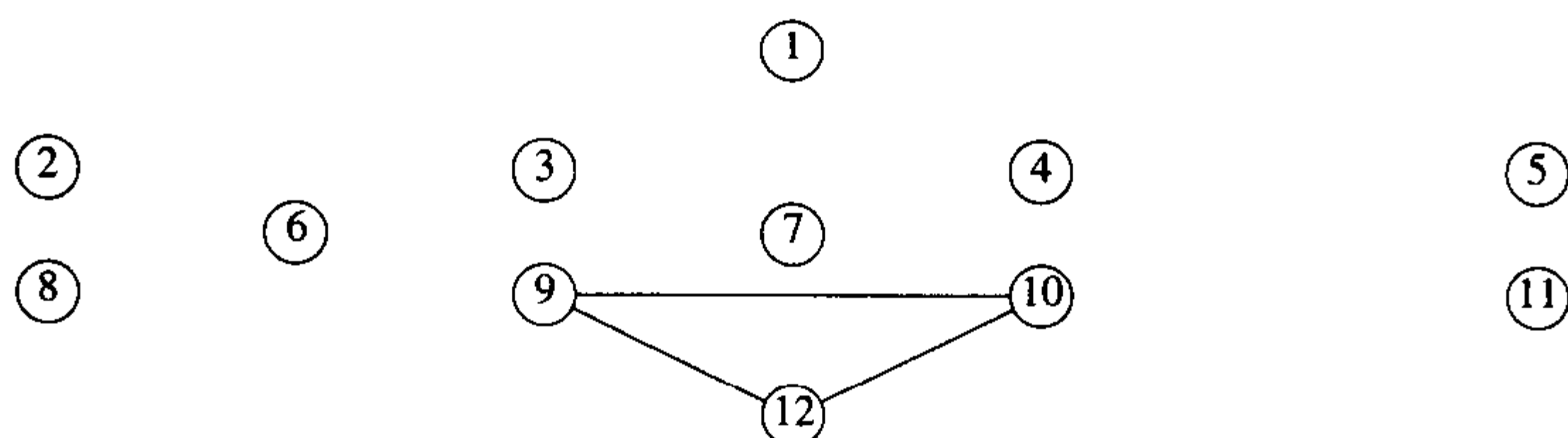


图9-73 遍历路径5、4、1、3、2、8、9、6、3、7、4、11、10、7、9、3、4、10、5后的图

为使算法更有效，必须使用适当的数据结构。我们将概述算法思想而把实现方法留做练习。  
 387 为使拼接简单，应该把路径作为一个链表保留。为避免重复扫描邻接表，对于每一个邻接表必须保留最后扫描到的边。当拼接进一个路径时，必须从拼接点开始搜索新顶点，从这个新顶点进行下一轮深度优先搜索。这将保证在整个算法期间对顶点搜索阶段所进行的全部工作量为 $O(|E|)$ 。使用适当的数据结构，算法的运行时间为 $O(|E|+|V|)$ 。

一个非常相似的问题是在无向图中寻找一个简单的回路，该回路通过图的每一个顶点。这个问题称为哈密尔顿回路问题 (Hamiltonian cycle problem)。虽然这个问题看起来和欧拉回路问题差不多，但是，对它却没有已知的有效算法。我们将在9.7节中再次看到这个问题。

#### 9.6.4 有向图

利用与无向图相同的思路，也可以通过深度优先搜索以线性时间遍历有向图。如果图不是强连通的，那么从某个结点开始的深度优先搜索可能访问不了所有的结点。在这种情况下我们从某个未作标记的结点处开始，反复执行深度优先搜索，直到所有的结点都被访问到。作为例子，考虑图9-74中的有向图。

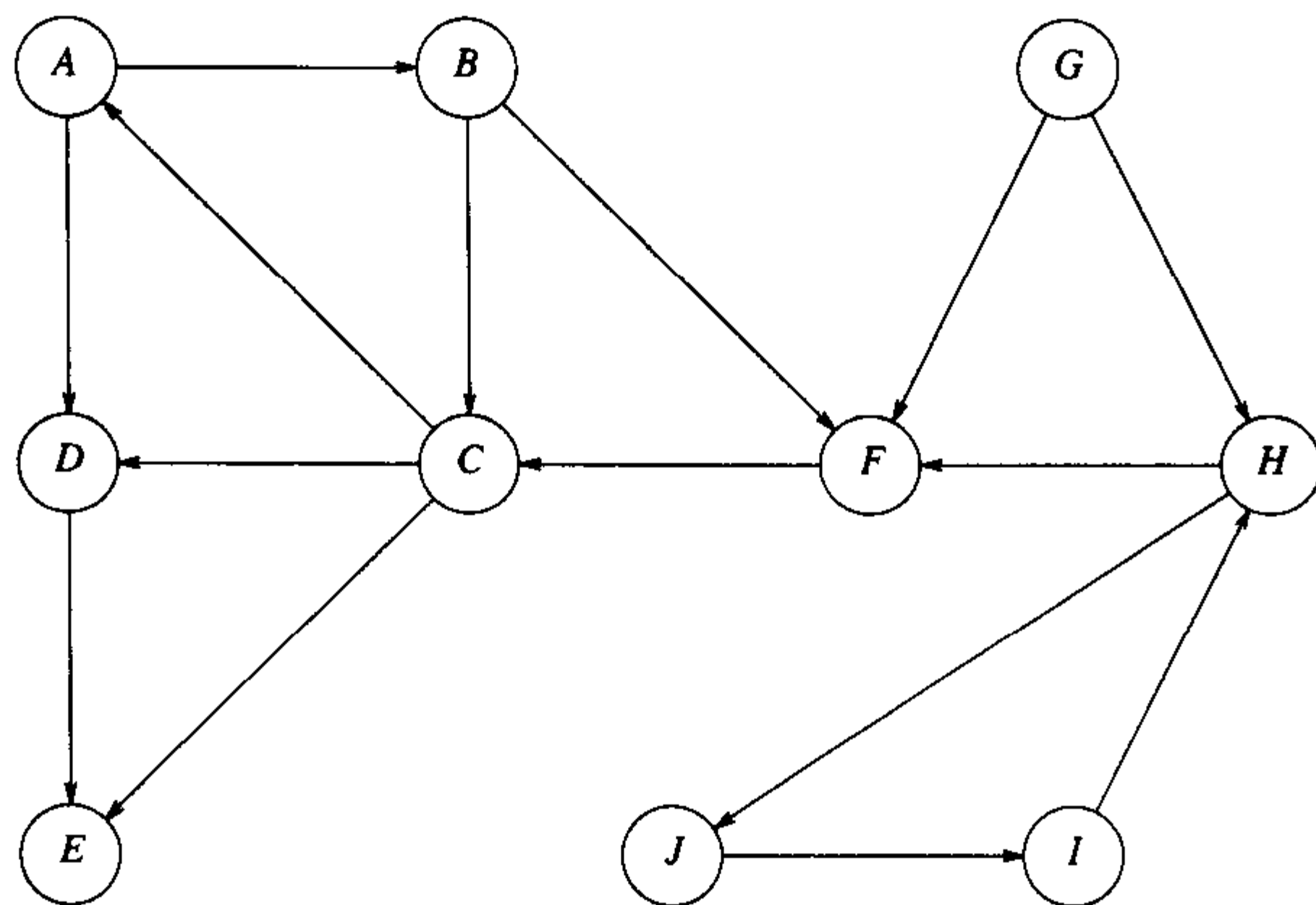


图9-74 一个有向图

我们从顶点 $B$ 任意地开始深度优先搜索。它访问顶点 $B$ 、 $C$ 、 $A$ 、 $D$ 、 $E$ 和 $F$ 。然后，从某个未访问的顶点重新开始。任意地，我们从 $H$ 开始，访问 $J$ 和 $I$ 。最后，从 $G$ 点开始，它是最后一个需要访问的顶点。对应的深度优先搜索树如图9-75所示。

388

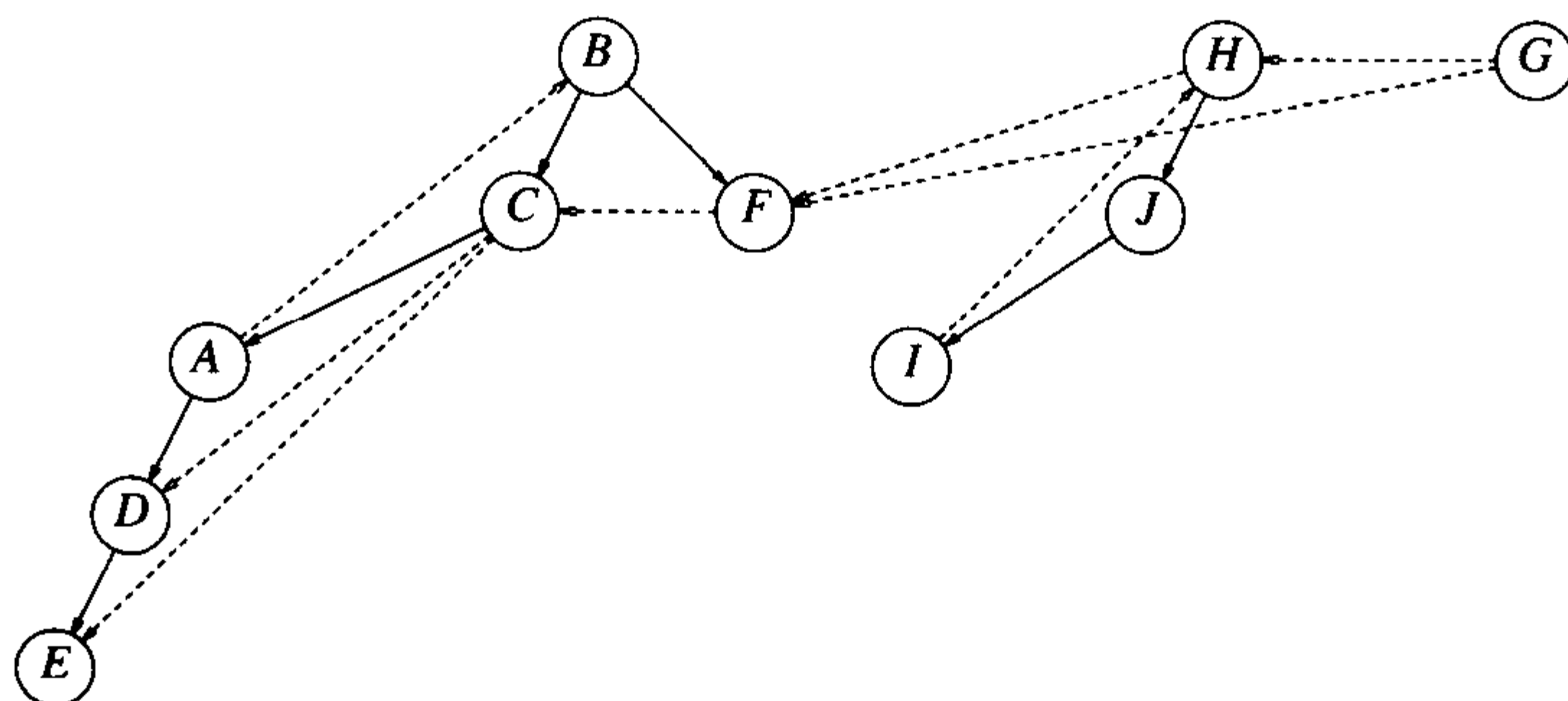


图9-75 图9-74的深度优先搜索

深度优先先生成森林中的虚线箭头是一些 $(v, w)$ 边，其中的 $w$ 在考察时已经做了标记。在无向图中，它们总是一些后向边，但是可以看到，存在三种类型的边并不通向新的顶点。首先是一些后向边 (back edge)，如 $(A, B)$ 和 $(I, H)$ 。还有一些前向边 (forward edge)，如 $(C, D)$ 和 $(C, E)$ ，它们从树的一个结点通向一个后裔。最后是一些交叉边 (cross edge)，如 $(F, C)$ 和 $(G, F)$ ，它们把不直接相关的两个树结点连接起来。深度优先搜索森林一般通过把一些子结点和一些新的树从左到右添加到森林中形成。在以这种方式画出的有向图的深度优先搜索中，交叉边总是从右到左行进的。

389

有些使用深度优先搜索的算法需要区别非树边的三种类型。当进行深度优先搜索时这是容易检验的，我们把它留作练习。

深度优先搜索的一种用途是检测一个有向图是否是无环图，法则如下：一个有向图是无环图当且仅当它没有后向边（上面的图有后向边，因此不是无环图）。读者可能还记得，拓扑排序也可以用来确定一个图是否是无环图。进行拓扑排序的另一种方法是通过深度优先先生成森林的后序遍历给顶点指定拓扑编号 $N, N-1, \dots, 1$ 。只要图是无环的，这种排序就是一致的。

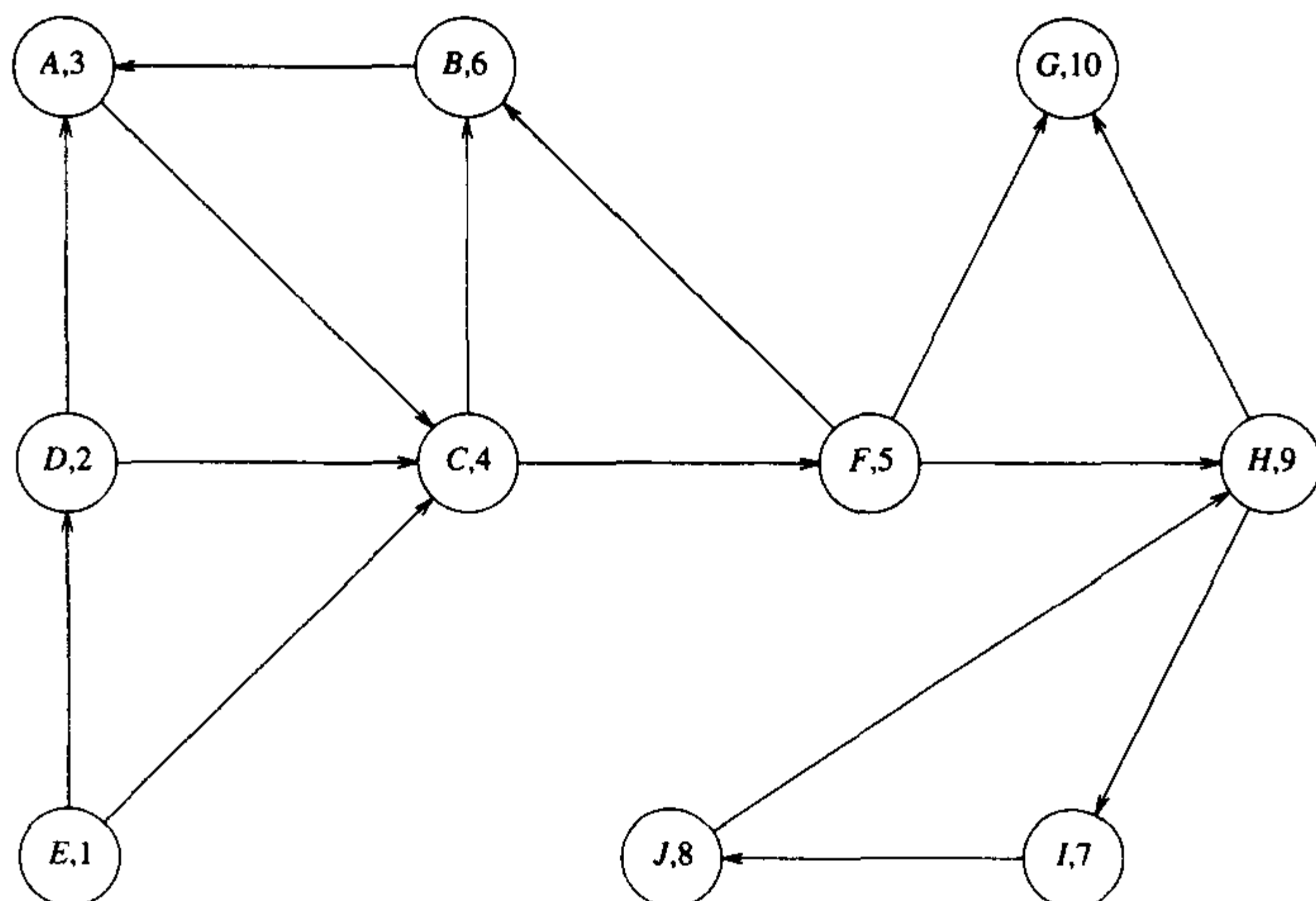
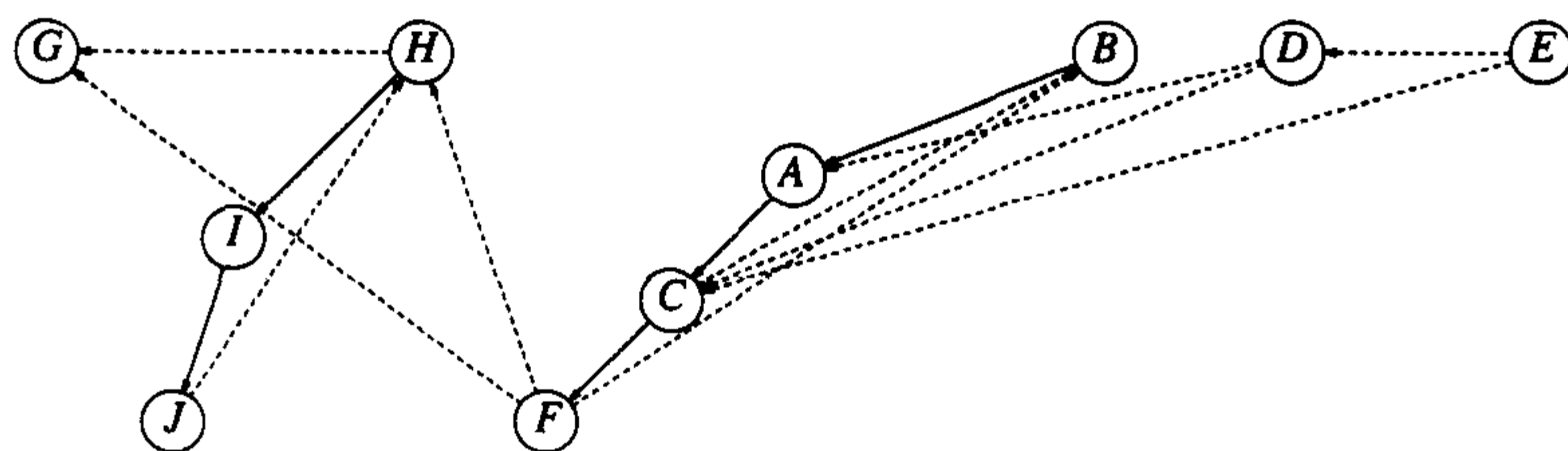
### 9.6.5 查找强分支

通过执行两次深度优先搜索，可以检测一个有向图是否是强连通的，如果它不是强连通的，那么实际上可以得到顶点的子集，它们到其自身是强连通的。这也可以只用一次深度优先搜索做到，不过，此处所使用的方法理解起来要简单得多。

首先，在一个输入的图 $G$ 上执行一次深度优先搜索。通过对深度优先先生成森林的后序遍历将 $G$ 的顶点编号，然后再把 $G$ 的所有的边反向，形成 $G_r$ 。图9-76代表图9-74所示的图 $G$ 的 $G_r$ ；顶点用它们的编号标出。

该算法通过对 $G_r$ 执行一次深度优先搜索而完成，总是在编号最高的顶点开始一次新的深度优先搜索。于是，从顶点 $G$ 开始对 $G_r$ 的深度优先搜索， $G$ 的编号为10。但该顶点不通向任何顶点，因此下一次搜索从 $H$ 点开始，这次调用访问 $I$ 和 $J$ 。下一次调用从 $B$ 点开始并访问 $A$ 、 $C$ 和 $F$ 。此后的调用是 $\text{dfs}(D)$ 及最终调用 $\text{dfs}(E)$ 。结果得到的深度优先先生成森林如图9-77所示。



图9-76 通过对(图9-74中的)图 $G$ 的后序遍历所编号的 $G_r$ 图9-77  $G_r$ 的深度优先搜索——强分支为 $\{G\}$ 、 $\{H, I, J\}$ 、 $\{B, A, C, F\}$ 、 $\{D\}$ 、 $\{E\}$ 

该深度优先生成森林中的每棵树(如果完全忽略所有的非树边,那么这是很容易看出的)形成一个强连通的分支。因此,对于该例子,这些强连通分支为 $\{G\}$ 、 $\{H, I, J\}$ 、 $\{B, A, C, F\}$ 、 $\{D\}$ 和 $\{E\}$ 。

为了理解该算法为什么成立,首先注意到,如果两个顶点 $v$ 和 $w$ 都在同一个强连通分支中,那么在原图 $G$ 中就存在从 $v$ 到 $w$ 的路径和从 $w$ 到 $v$ 的路径,因此,在 $G_r$ 中也存在。现在,如果两个顶点 $v$ 和 $w$ 不在 $G_r$ 的同一个深度优先生成树中,那么显然它们也不可能在同一个强连通分支中。

为了证明该算法成立,必须指出,如果两个顶点 $v$ 和 $w$ 在 $G_r$ 的同一个深度优先生成树中,那么必然存在从 $v$ 到 $w$ 的路径和从 $w$ 到 $v$ 的路径。等价地,可以证明,如果 $x$ 是 $G_r$ 包含 $v$ 的深度优先生成树的根,那么存在一条从 $x$ 到 $v$ 和从 $v$ 到 $x$ 的路径。对 $w$ 应用相同的推理,则得到一条从 $x$ 到 $w$ 和从 $w$ 到 $x$ 的路径。这些路径则意味着那些从 $v$ 到 $w$ 和从 $w$ 到 $v$ (经过 $x$ )的路径。

由于 $v$ 是 $x$ 在 $G_r$ 的深度优先生成树中的一个后裔,因此 $G_r$ 中存在一条从 $x$ 到 $v$ 的路径,从而 $G$ 中存在一条从 $v$ 到 $x$ 的路径。此外,由于 $x$ 是根结点,因此 $x$ 从第一次深度优先搜索得到更高的后序编号。于是,在第一次深度优先搜索期间所有处理 $v$ 的工作都在 $x$ 的工作结束前完成。既然存在一条从 $v$ 到 $x$ 的路径, $v$ 必然是 $x$ 在 $G$ 的生成树中的一个后裔——否则 $v$ 将在 $x$ 之后结束。这意味着 $G$ 中有一条从 $x$ 到 $v$ 的路径,证明结束。

## 9.7 NP完全性介绍

在这一章,我们已经看到各种各样图论问题的解法。所有这些问题都有一个多项式运行时间,除网络流问题外,运行时间或者是线性的,或者比线性稍微多一些( $O(|E|\log|E|)$ )。顺便指出,

我们还提到，对于某些问题，有些变化似乎比原问题要难。

回忆欧拉回路问题，它要求找出一条恰好经过图的每条边一次的路径，该问题是线性时间可解的。哈密尔顿回路问题要找一个简单回路，该回路包含图的每一个顶点。对于这个问题，尚不知道有线性算法。

对于有向图的单源无权最短路径问题也是线性时间可解的。但对应的最长简单路径问题尚不知有线性时间算法。

这些问题的变种，其情况实际上比我们描述的还要糟。对于这些变种问题不仅不知道线性算法，而且不存在保证以多项式时间运行的已知算法。这些问题的一些熟知算法对于某些输入可能要花费指数时间。

在这一节，我们将简要考察这种问题。这种问题是相当复杂的，因此这里只进行快速和非正式的探讨。这样一来，我们的讨论可能（必然地）处处都或多或少地有些不准确。

我们将看到，存在大量重要的问题，它们在复杂性上大体是等价的。这些问题形成一个类，叫作**NP完全**（NP-complete）问题。这些NP完全问题精确的复杂度仍然需要确定并且在计算机科学理论方面仍然是最重要的开放性问题。或者所有这些问题都有多项式时间解法，或者它们都没有多项式时间解法。

### 9.7.1 难与易

在给问题分类时，第一步要考虑的是分界。我们已经看到，许多问题可以用线性时间求解。我们还看到某些 $O(\log N)$ 的运行时间，但是它们或者假定已进行过某些预处理（如输入数据已读入或数据结构已建立），或者出现在运算实例中。例如，gcd（最大公因数）算法，当用于两个数 $M$ 和 $N$ 时，花费 $O(\log N)$ 时间。由于这两个数分别由 $\log M$ 和 $\log N$ 个二进制位组成，因此gcd算法实际上花费的时间对于输入数据的量或大小而言是线性的。由此可知，当度量运行时间时，我们将把运行时间考虑成输入数据的量的函数。一般说来，我们不能期望运行时间比线性更好。

另一方面，确实存在某些真正难的问题。这些问题非常难，以至于不可能解出。但这并不意味着只能发出叹息，期待天才来求解该问题。正如实数不足以表示 $x^2 < 0$ 的解一样，可以证明，计算机不可能解决碰巧发生的每一个问题。这些“不可能”解出的问题称为**不可判定问题**（undecidable problem）。

一个特殊的不可判定问题是**停机问题**（halting problem）。是否能够让C++编译器拥有一个附加的特性，即不仅能够检查语法错误，而且还能够检查所有的无限循环？这似乎是一个难的问题，但是我们或许期望，假如某些非常聪明的程序员花上足够的时间，他们也许能够编制出这种增强型的编译器。

该问题是不可判定问题的直观原因在于，这样一个程序可能很难检查它自己。由于这个原因，有时称这些问题为**递归不可判定的**（recursively undecidable）。

如果能够写出一个无限循环检查程序，那么它肯定可以用于自检。假设可以编制出一个程序叫作LOOP。LOOP把一个程序 $P$ 作为输入并使 $P$ 自身运行。如果 $P$ 自身运行时出现循环，则显示短语YES。如果 $P$ 自身运行时终止了，那么自然是显示NO。现在不这样做，而是让LOOP进入一个无限循环。

当LOOP将自身作为输入时会发生什么呢？LOOP或者停止，或者不停止。问题在于，这两种可能性均导致矛盾，与短语“这句话是谎言”产生的矛盾大致相同。

根据我们的定义，如果 $P(P)$ 终止，则 $LOOP(P)$ 进入一个无限循环。设当 $P = LOOP$ 时， $P(P)$ 终止。此时，按照LOOP程序， $LOOP(P)$ 应该进入一个无限循环。因此，必须让 $LOOP(LOOP)$ 终

止并进入一个无限循环，显然这是不可能的。另一方面，设当 $P = LOOP$ 时， $P(P)$ 进入一个无限循环，则 $LOOP(P)$ 必然终止，而我们得到同样的一组矛盾。因此，可以看到，程序 $LOOP$ 不可能存在。

### 9.7.2 NP类

NP类是在难度上稍逊于不可判定问题的类。NP代表非确定型多项式时间（nondeterministic polynomial-time）。确定型机器在每一时刻都在执行一条指令。根据这条指令，机器再去执行某条接下来的指令，这是唯一确定的。而非确定型机器对其后的步骤是有选择的。它可以自由进行它想要的任意的选择，如果这些后面的步骤中有一条导致问题的解，那么它将总是选择这个正确的步骤。因此，非确定型机器具有非常好的猜测（优化）能力。这就好像一个奇怪的模型，因为没有人能够构建一台非确定型计算机，还因为这台机器是对标准计算机的令人难以置信的改进（此时每一个问题都变成易解的了）。我们将看到，非确定性是非常有用的理论结构。此外，非确定性也不像人们想像中那么强大。例如，即使使用非确定性，不可判定问题仍然还是不可判定的。

检验一个问题是否属于NP的简单方法是将该问题用“是/否（yes/no）问题”的语言描述。如果我们在多项式时间内能够证明一个问题的任意“是”的实例是正确的，那么该问题就属于NP类。我们不必担心“否”的实例，因为程序总是进行正确的选择。因此，对于哈密尔顿回路问题，一个“是”的实例就是图中任意一个包含所有顶点的简单的回路。由于给定一条路径，验证它是否真的是哈密尔顿回路是一件简单的事情，因此哈密尔顿回路问题属于NP。诸如“存在长度大于 $K$ 的简单路径吗？”这样的适当的问题也可能容易验证，从而属于NP。满足这条性质的任何路径均可容易地检验。

393

由于解本身显然提供了验证方法，因此，NP类包括所有具有多项式时间解的问题。人们会想到，既然验证一个答案要比经过计算提出一个答案容易得多，因此在NP中就会存在不具有多项式时间解法的问题。这样的问题至今没有发现，于是，非确定性并不是如此重要的改进，这是完全有可能的，尽管有些专家很可能不这么认为。问题在于，证明指数下界是一项极其困难的工作。我们曾用来证明排序需要 $\Omega(M \log N)$ 次比较的信息论定界方法似乎还不足以完成这样的工作，因为决策树都远不够大。

还要注意，不是所有的可判定问题都属于NP。考虑确定一个图是否没有哈密尔顿回路的问题。证明一个图有哈密尔顿回路是相对简单的一件事情——只需展示一个即可。然而却没有人知道如何以多项式时间证明一个图没有哈密尔顿回路。似乎人们只能枚举所有的回路并且一个一个地验证它们才行。因此，无哈密尔顿回路的问题不属于NP。

### 9.7.3 NP完全问题

在已知属于NP的所有问题中，存在一个子集，叫作NP完全（NP-complete）问题，它包含了NP中最难的问题。NP完全问题有一个性质，即NP中的任一问题都能够多项式地归约（polynomially reduced）成NP完全问题。

一个问题 $P_1$ 可以如下归约成问题 $P_2$ ：设有一个映射，使得 $P_1$ 的任何实例都可以变换成 $P_2$ 的一个实例。求解 $P_2$ ，然后将答案映射回原始的解答。作为一个例子，考虑把数以十进制输入到计算器。将十进制数转化成二进制数，所有的计算都用二进制进行。然后，再把最后的答案转变成十进制显示。对于可多项式地归约成 $P_2$ 的 $P_1$ ，与变换相联系的所有工作必然以多项式时间完成。

NP完全问题是最难的NP问题的原因在于，一个NP完全问题基本上可以用作NP中任何问题的子例程，其花费只不过是多项式的开销量。因此，如果任意NP完全问题有一个多项式时间解，



那么NP中的每一个问题必然都有一个多项式时间的解。这使得NP完全问题是所有NP问题中最难的问题。

设我们有一个NP完全问题 $P_1$ ，并设 $P_2$ 已知属于NP。再进一步假设 $P_1$ 多项式地归约成 $P_2$ ，使得我们可以通过使用 $P_2$ 求解 $P_1$ 而只多损耗了多项式时间。由于 $P_1$ 是NP完全的，所以NP中的每一个问题都可多项式地归约成 $P_1$ 。应用多项式的封闭性，可以看到，NP中的每一个问题均可多项式地归约成 $P_2$ ：把问题归约成 $P_1$ ，然后再把 $P_1$ 归约成 $P_2$ 。因此， $P_2$ 是NP完全的。

作为一个例子，设我们已经知道哈密尔顿回路问题是NP完全问题。旅行商问题（traveling salesman problem）如下。

**旅行商问题** 给定一完全图 $G=(V, E)$ 、其边的值以及整数 $K$ ，是否存在一个访问所有顶点并且总值小于等于 $K$ 的简单回路？

这个问题不同于哈密尔顿回路问题，因为全部 $|V|(|V|-1)/2$ 条边都存在而且图是加权图。该问题有很多重要的应用。例如，印刷电路板需要穿一些孔使得芯片、电阻器以及其他的电子元件可以置入，这是可以机械完成的。穿孔是快速的操作；时间耗费在给穿孔器定位上，定位所需要的时间依赖于从孔到孔间行进的距离。由于我们希望给每一个孔位穿孔（然后返回到开始位置以便给下一块电路板穿孔），并将钻头移动所耗费的总时间限制到最小，因此得到的是一个旅行商问题。

旅行商问题是NP完全的。容易看到，其解可以用多项式时间检验，当然它属于NP。为了证明它是NP完全的，可多项式地将哈密尔顿回路问题归约为旅行商问题。为此，构造一个新的图 $G'$ ， $G'$ 和 $G$ 有相同的顶点。对于 $G'$ 的每一条边 $(v, w)$ ，如果 $(v, w) \in G$ ，那么它就有权1，否则，它的权就是2。我们选取 $K=|V|$ 。见图9-78。

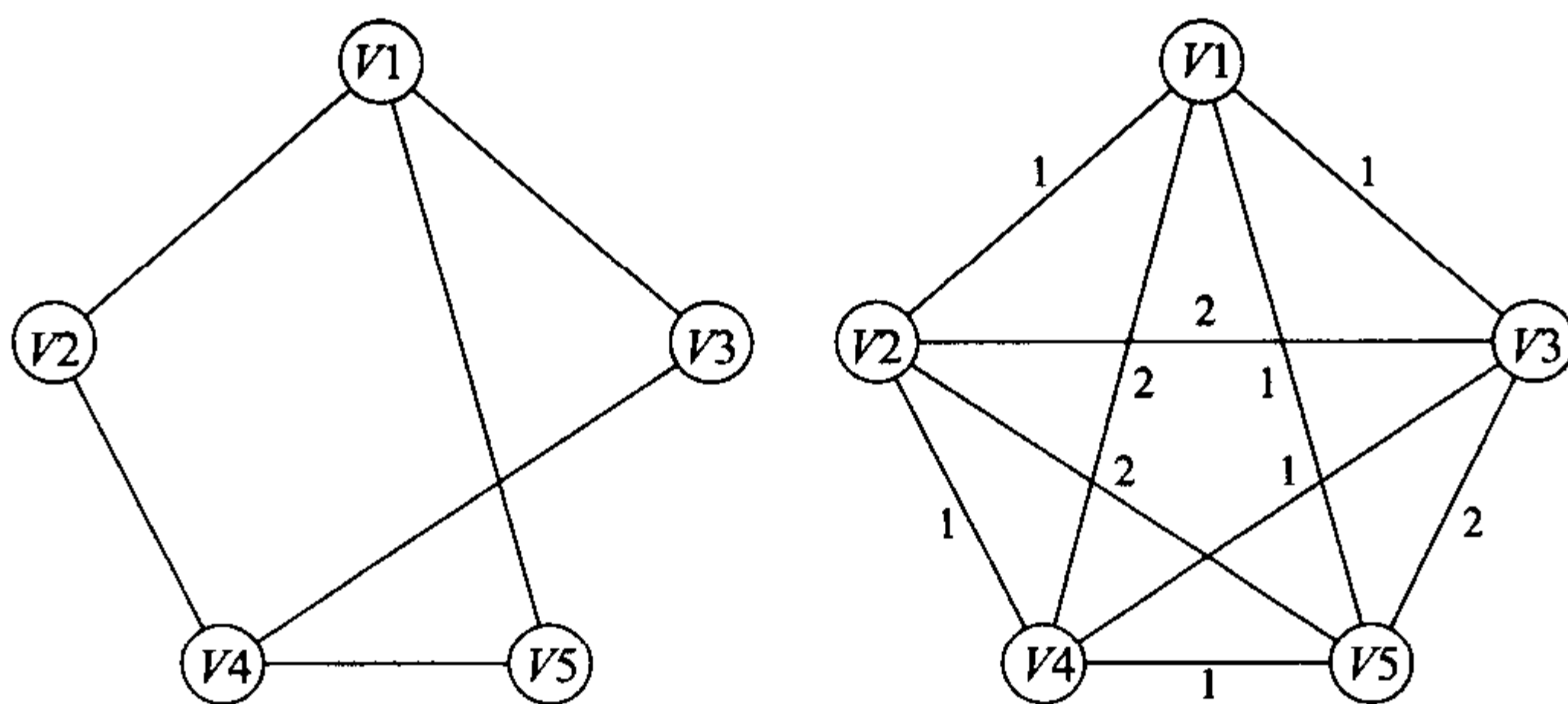


图9-78 哈密尔顿回路问题变换成旅行商问题

容易验证， $G$ 有一个哈密尔顿回路当且仅当 $G'$ 有一个总权为 $|V|$ 的旅行商的巡回路线。

现在有许多已知是NP完全的问题。为了证明某个新问题是NP完全的，必须证明它属于NP，然后将一个适当的NP完全问题变换到该问题。虽然到旅行商问题的变换是相当简单的，但是，大部分变换实际上却是相当复杂的，需要某些复杂的构造。一般来说，在考虑了多个不同的NP完全问题之后才考虑实际提供归约的问题。由于我们只关注一般的想法，因此也就不再讨论更多的变换；有兴趣的读者可以查阅本章后面的参考文献。

细心的读者可能想知道第一个NP完全问题是如何具体地被证明是NP完全的。由于证明一个问题是NP完全的需要从另外一个NP完全问题变换到它，因此必然存在某个NP完全问题，对于这个问题不能使用上述的思路。第一个被证明是NP完全的问题是可满足性（satisfiability）问题。这个可满足性问题把一个布尔表达式作为输入，并提问是否该表达式对式中各变量的一次赋值取



395 值true。

可满足性当然属于NP，因为很容易计算一个布尔表达式的值并检查结果是否为真（true）。在1971年，Cook通过直接证明NP中的所有问题都可以变换成可满足性问题而证明了可满足性问题是NP完全的。为此，他用到了对NP中每一个问题都已知的事实：NP中的每一个问题都可以用一台非确定型计算机在多项式时间内求解。计算机的一个形式化的模型称为图灵机（Turing machine）。Cook指出这台机器的动作如何能够用一个极其复杂但仍然是多项式的冗长的布尔公式来模拟。该布尔公式为真，当且仅当由图灵机运行的程序对其输入得到一个“是”的答案。

一旦可满足性被证明是NP完全的，则一大批新的NP完全问题，包括某些最经典的问题，也都被证明是NP完全的。

除可满足性问题外，我们已经考查过的哈密尔顿回路问题、旅行商问题、最长路径问题，都是NP完全问题。此外，还有一些我们尚未讨论的问题，如装箱（bin packing）问题、背包（knapsack）问题、图的着色（graph coloring）问题以及团（clique）问题都是著名的NP完全问题。NP完全问题相当广泛，包括来自操作系统（调度和安全）、数据库系统、运筹学、逻辑学特别是图论等不同领域的问题。

## 小结

在本章中，我们已经看到图如何用来给出许多实际生活问题的模型。实际出现的图常常是非常稀疏的，因此，注意用于实现这些图的数据结构很重要。

我们还看到一类问题，它们似乎没有有效的解法。在第10章将讨论处理这些问题的某些方法。

## 练习

396 9.1 找出图9-79中的一个拓扑排序。

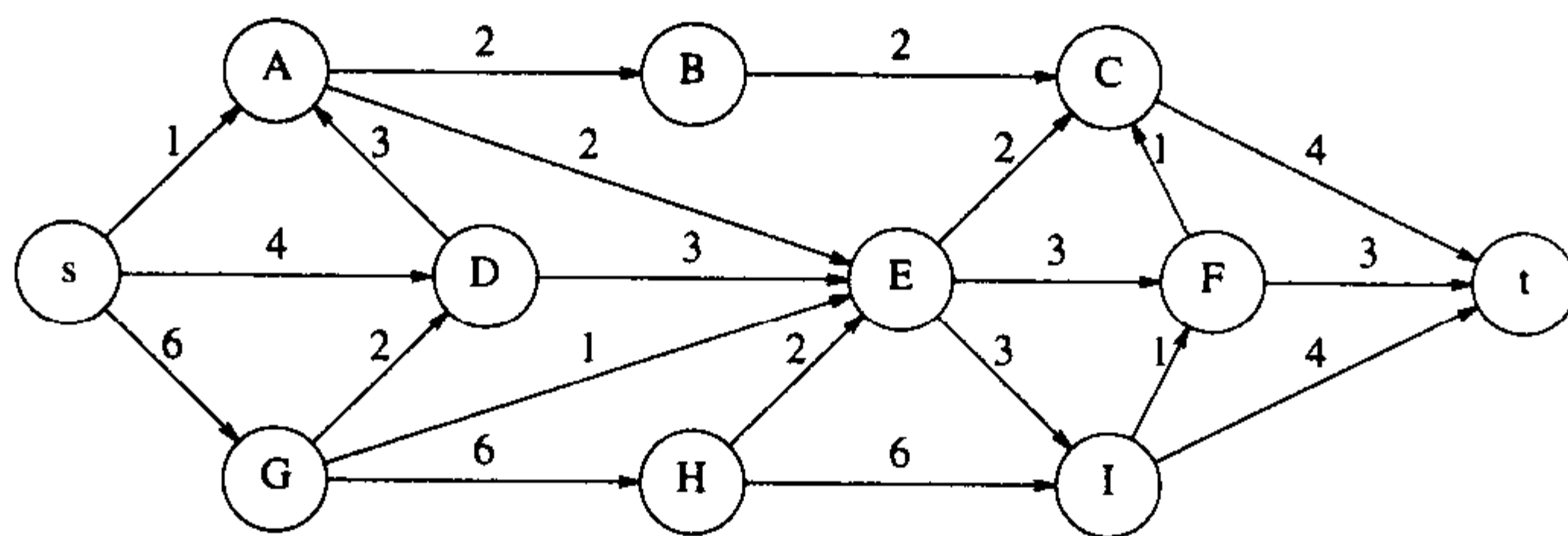


图9-79 练习9.1和练习9.11所使用的图

- 9.2 如果用一个栈代替9.2节中拓扑排序算法的队列，是否得到不同的排序？为什么一种数据结构会给出“更好”的答案？
- 9.3 编写一个程序实现对图的拓扑排序。
- 9.4 使用标准的二重循环，一个邻接矩阵仅仅初始化就需要 $O(|V|^2)$ 时间。试提出一种方法将图存储在一个邻接矩阵中（使得测试一条边是否存在花费 $O(1)$ ），但避免二次的运行时间。
- 9.5
  - a. 找出图9-80中A点到所有其他顶点的最短路径。
  - b. 找出图9-80中B点到所有其他顶点的最短无权路径。

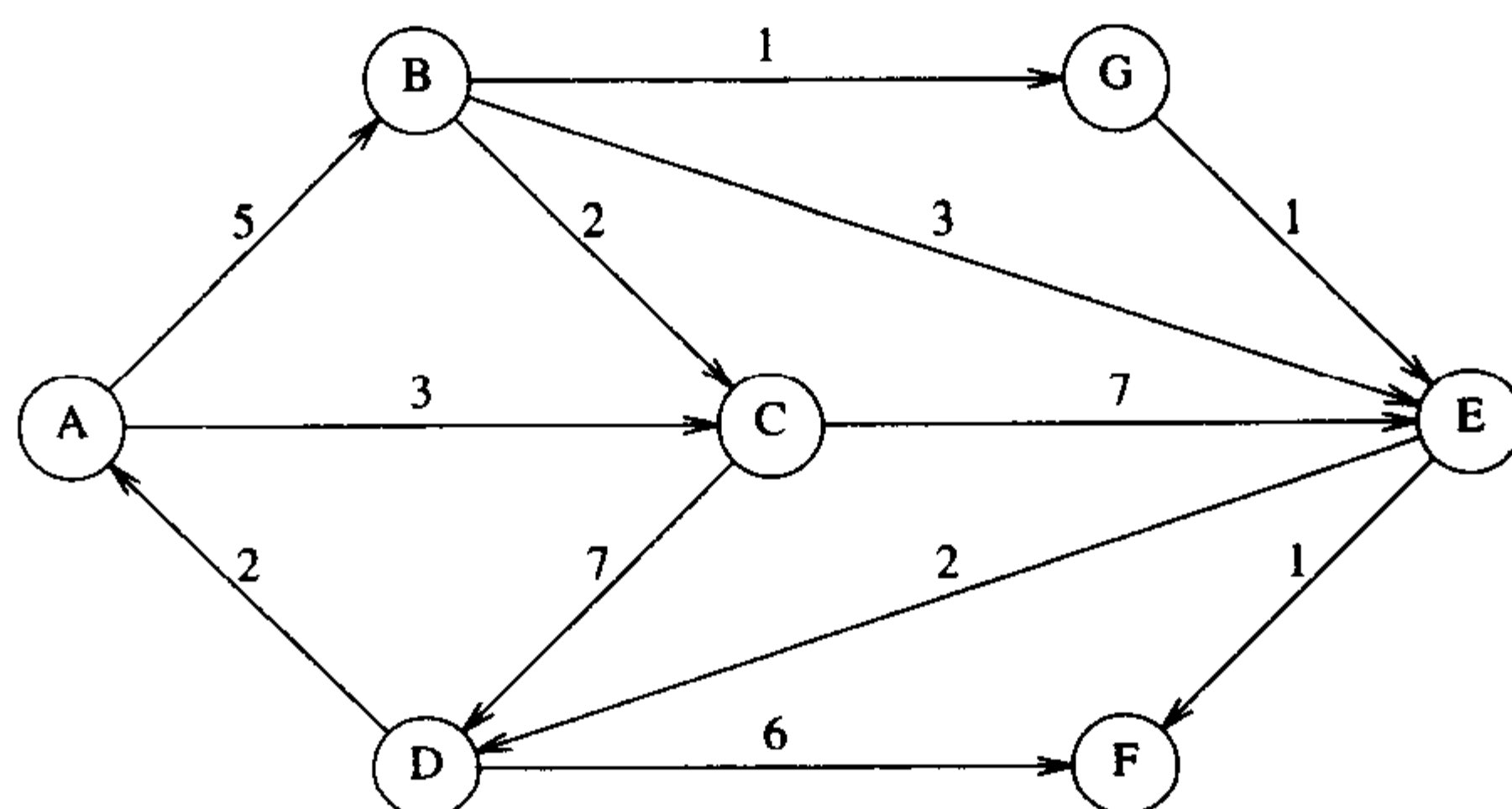


图9-80 练习9.5使用的图

- 9.6 当用 $d$ 堆实现时（见6.5节），Dijkstra算法最坏情形的运行时间是多少？
- 9.7 a. 给出在有一条负边但无负值回路时Dijkstra算法得到错误答案的例子。  
 \*\* b. 证明：如果存在负权边但无负值回路，则9.3.3节中提出的加权最短路径算法是成立的，并证明该算法的运行时间为 $O(|E| |V|)$ 。
- \*9.8 设一个图的所有边的权都是1和 $|E|$ 之间的整数。Dijkstra算法可以多快实现？
- 9.9 写出一个程序来求解单源最短路径问题。
- 9.10 a. 解释如何修改Dijkstra算法以得到从 $v$ 到 $w$ 的不同的最小路径的个数。  
 b. 解释如何修改Dijkstra算法，使得如果存在多条从 $v$ 到 $w$ 的最小路径，那么具有最少边数的路径将被选中。
- 9.11 找出图9-79中的网络的最大流。
- 9.12 设 $G = (V, E)$ 是一棵树， $s$ 是它的根，并且添加一个顶点 $t$ 以及从 $G$ 中所有树叶到 $t$ 的无穷容量的边。给出一个线性时间算法以找出从 $s$ 到 $t$ 的最大流。
- 9.13  $G = (V, E)$ 是把 $V$ 划分成两个子集 $V_1$ 和 $V_2$ 并且其边的两个顶点都不在同一个子集中的二分图。  
 a. 给出一个线性算法以确定一个图是否是二分图。  
 b. 二分匹配问题是找出 $E$ 的最大子集 $E'$ 使得没有顶点含在多条边中。图9-81中所示的是四条边的一个匹配（用虚线表示）。存在一个五条边的匹配，它是最大的匹配。  
 c. 指出二分匹配问题如何用于解决下列问题：我们有一组教师、一组课程以及每位教师有资格教授的课程表。如果教师不需要教授多门的课程，而且只有一位教师可以教授一门给定的课程，那么可以开设的课程最多是多少？  
 d. 证明网络流问题可以用来解决二分匹配问题。  
 e. 问题b的解法的时间复杂度如何？

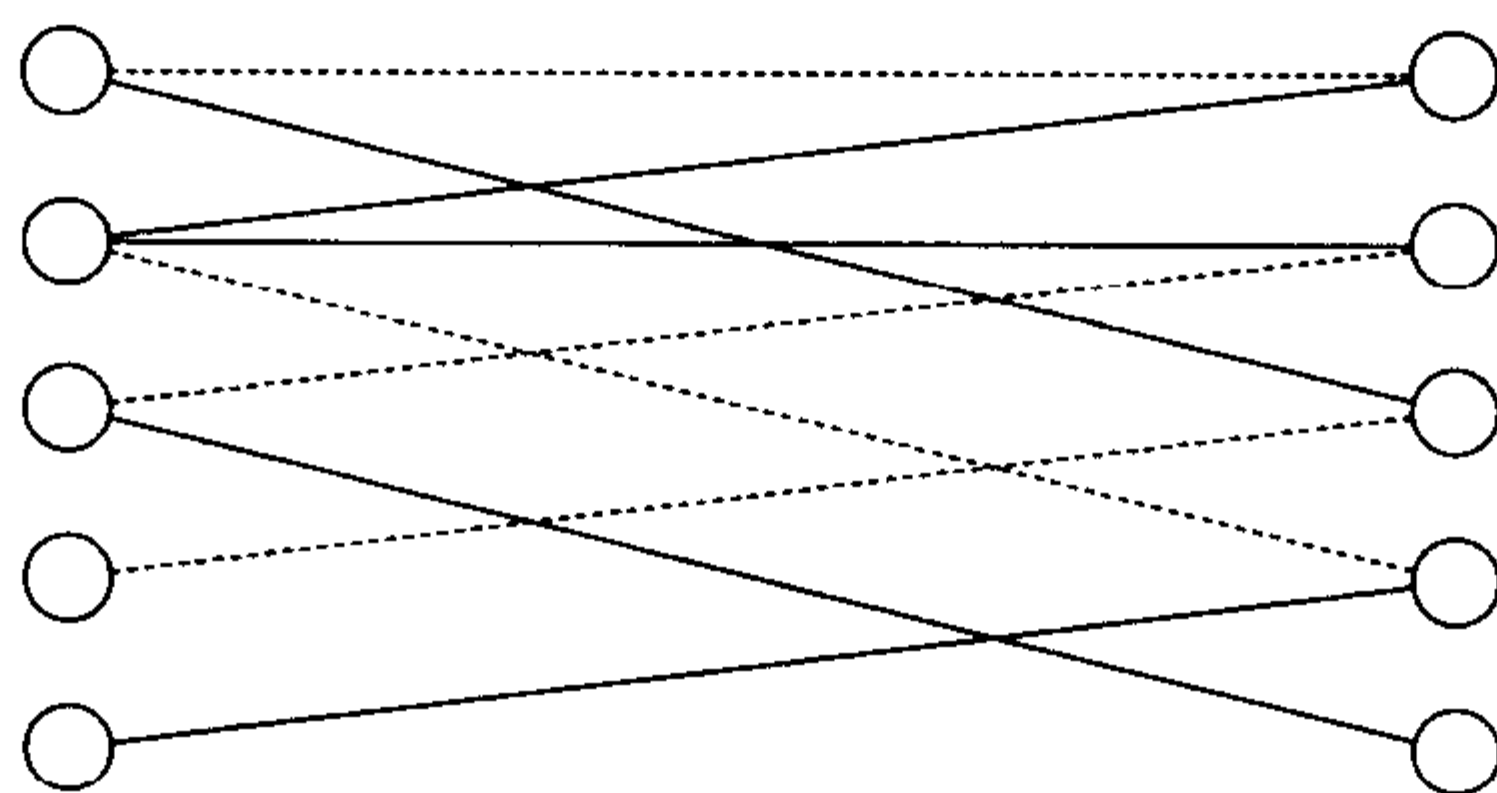


图9-81 一个二分图

- 9.14 给出一个算法，找出容许最大流通过的增长路径。
- 9.15 a. 使用Prim和Kruskal两种算法求出图9-82的最小生成树。  
 b. 这棵最小生成树是唯一的吗？为什么？

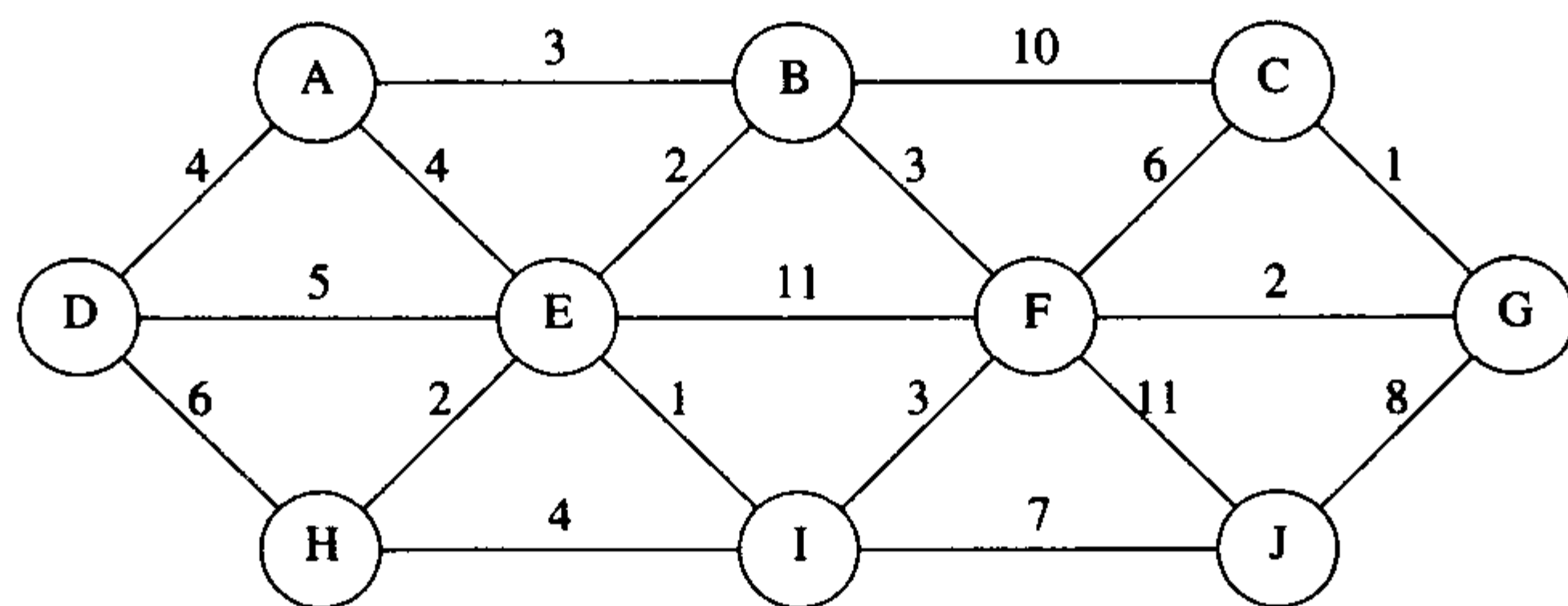


图9-82 练习9.15使用的图

- 9.16 如果有一些负的边权，那么Prim算法或Kruskal算法还能行得通吗？
- 9.17 证明 $V$ 个顶点的图可以有 $V^{V-2}$ 棵最小生成树。
- 9.18 编写一个程序实现Kruskal算法。
- 9.19 如果一个图的所有边的权都在1和 $|E|$ 之间，那么能有多快算出最小生成树？
- 9.20 给出一个算法求解最大生成树。这比求解最小生成树更难吗？
- 9.21 求出图9-83的所有割点。指出深度优先生成树和每个顶点的Num和Low的值。

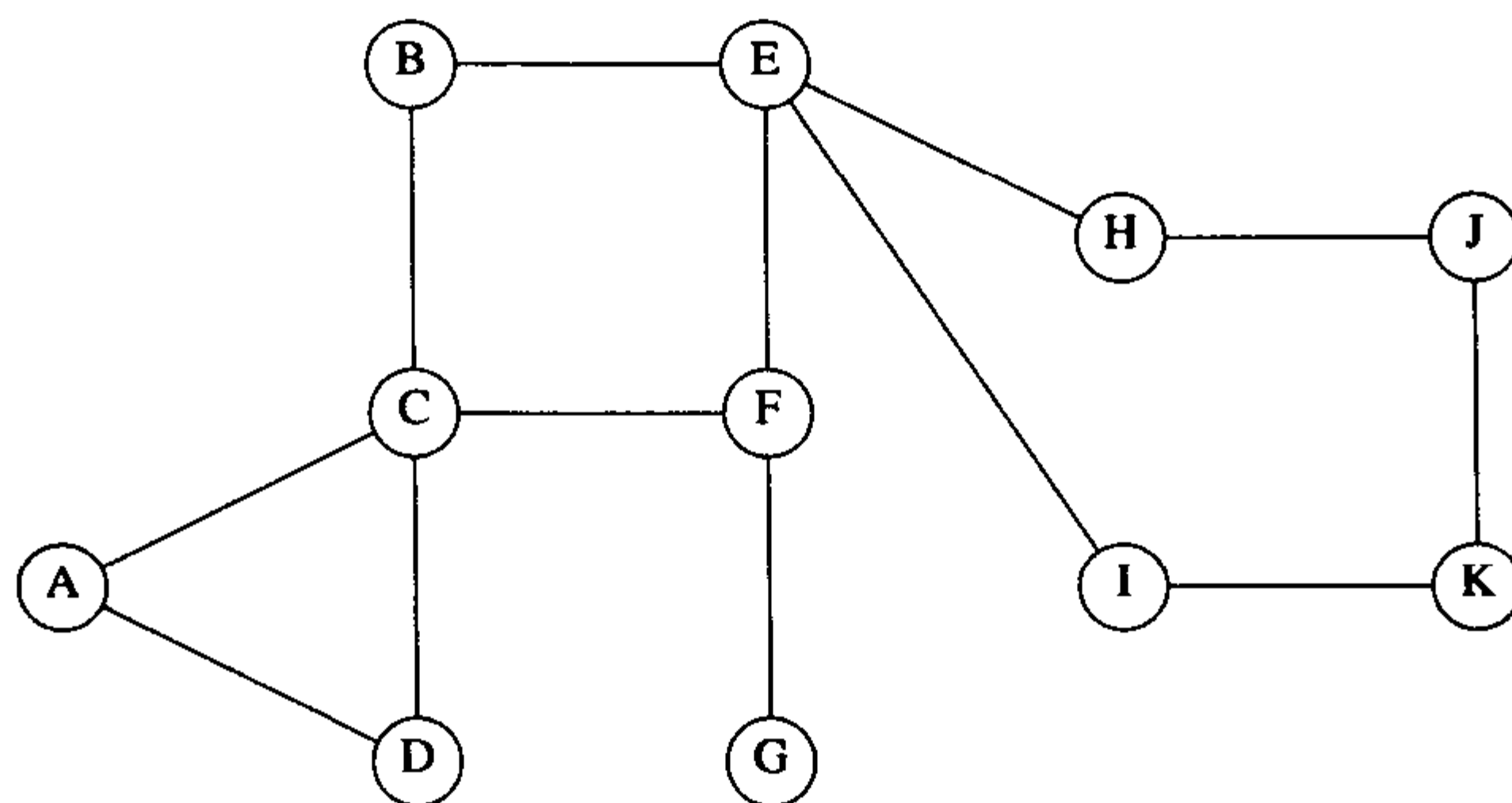


图9-83 练习9.21使用的图

- 9.22 证明查找割点的算法的正确性。
- 9.23 a. 给出一种算法，求出从无向图中删除后使得图是无环图所需要的最小的边数。  
\*b. 证明这个问题对有向图是NP完全的。
- 9.24 证明：在一个有向图的深度优先生成森林中所有的交叉边都是从右到左的。
- 9.25 给出一个算法以决定在一个有向图的深度优先生成森林中的一条边 $(v, w)$ 是否是树、后向边、交叉边或前向边。
- 9.26 找出图9-84中的强连通分支。

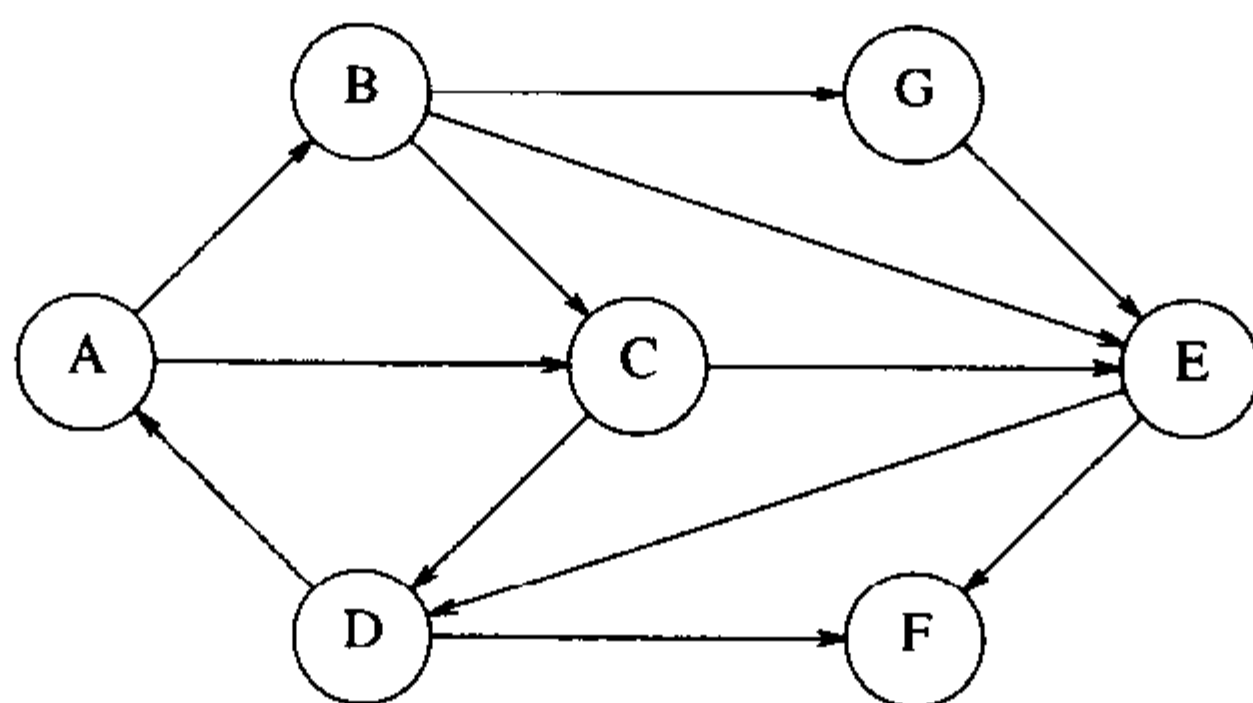


图9-84 练习9.26使用的图

- 9.27 编写一个程序，使它能找出有向图中的强连通分支。
- \*9.28 给出一个算法，只用一次深度优先搜索即可找出强连通分支来。使用类似于双连通性算法的算法。

- 9.29 图 $G$ 的双连通分支是把边划分成一些集合,使得每个边集所形成的图是双连通的。修改图9-67中的算法,使之能找出双连通分支而不是割点。
- 9.30 设我们对一个无向图进行广度优先搜索并建立一棵广度优先生成树。证明该树所有的边或者是树边或者是交叉边。
- 9.31 给出一个算法在无向(连通)图中找出一条路径,使其在每个方向上恰好通过每条边一次。
- 9.32 a. 编写一个程序以找出图中的一条欧拉回路(如果存在的话)。  
b. 编写一个程序以找出图中的一条欧拉环游(如果存在的话)。
- 9.33 有向图中的欧拉回路是一个回路,该回路中的每条边恰好被访问一次。  
\*a. 证明,有向图有欧拉回路当且仅当它是强连通的并且每个顶点的入度等于出度。  
\*b. 给出一个线性时间算法,在存在欧拉回路的有向图中找出一条欧拉回路。
- 9.34 a. 考虑欧拉回路问题的下列解法:假设一个图是双连通的。执行一次深度优先搜索,只在万不得已的时候使用后向边。如果图不是双连通的,则对双连通分支递归地应用该算法。这个算法行得通吗?  
b. 设当用到后向边时我们取用连接到最近祖先结点的后向边,该算法是否行得通?
- 9.35 平面图是可以画在平面上而其任何两条边都不相交的图。  
\*a. 证明:图9-85中的两个图都不是平面图。  
b. 证明:在平面图中必然存在某个顶点与最多不超过五个顶点相连。  
\*\*c. 证明:在平面图中 $|E| \leq 3|V| - 6$ 。

400

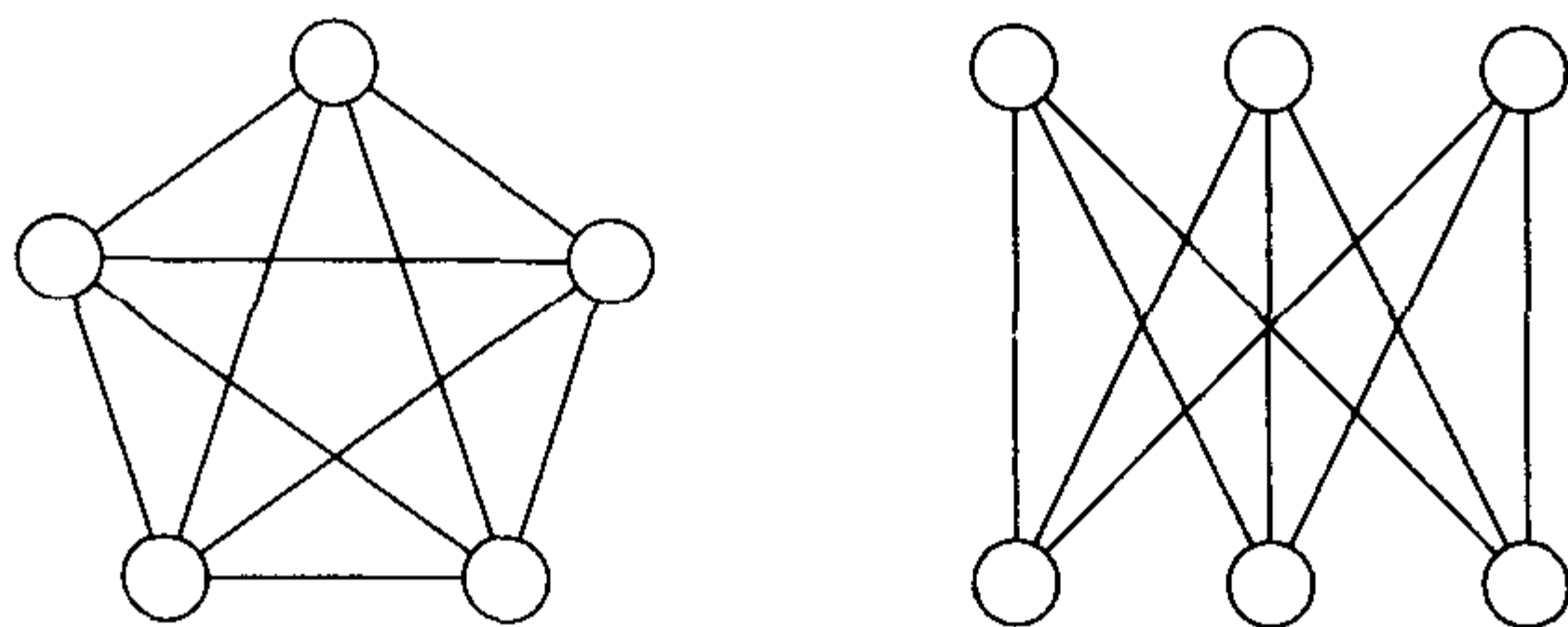


图9-85 练习9.35使用的图

- 9.36 多重图(multigraph)是其顶点对之间可以有多种边(multiple edge)的图。本章中哪些算法对于多重图不用修改就能正确运行?对其余的算法需要进行哪些修改?
- \*9.37 令 $G = (V, E)$ 是一个无向图。使用深度优先搜索设计一个线性算法,把 $G$ 的每条边转换成有向边使得所得到的图是强连通的,或者确定这是不可能的。
- 9.38 有 $N$ 根木棍,它们以某种结构相互叠压平放。每根木棍由它的两个端点确定;每个端点是由 $x$ 、 $y$ 和 $z$ 坐标确定的有序三元组;没有木棍垂直摆放。仅当木棍上没有其他木棍放置时才可以将其取走。  
a. 解释如何编写一个例程接收两根木棍 $a$ 和 $b$ 并报告 $a$ 是否在 $b$ 上面、 $b$ 下面或是与 $b$ 无关(本问与图论毫无关系)。  
b. 给出一个算法确定是否能够取走所有的木棍,如果能,那么提供完成这项工作的木棍拾取次序。
- 9.39 如果一个图的每个顶点都可以给定 $k$ 种颜色之一,并且没有边连接相同颜色的顶点,则称该图是 $k$ 可着色的。给出一个线性时间算法测试图的2可着色性。假设图以邻接表的形式存储;你必须指明任何所需要的附加的数据结构。
- 9.40 给出一种多项式时间算法,使在任意的无向图中能够找出个顶点,这些顶点至少覆盖图的 $3/4$ 的边。
- 9.41 指出如何修改拓扑排序算法,使得如果图不是无环图,则该算法将显示出某个回路来。可以不用深度优先搜索。
- 9.42 令 $G$ 为一有向图,该图有 $N$ 个顶点。如果对 $V$ 中每一个顶点 $v$ 有 $s \neq v$ ,且存在边 $(v, s)$ ,但是不存在



形如  $(s, v)$  的边，则顶点  $s$  叫作汇点。给出一个  $O(N)$  时间算法，确定  $G$  是否有汇点，假设  $G$  由  $N \times N$  邻接矩阵给定。

9.43 当把一个顶点和与它关联的边从树中除去后，则剩下一些子树。给出一个线性时间算法，来找出一个顶点，从  $N$  个顶点的树中删除该顶点将不会留下多于  $N/2$  个顶点的子树。

9.44 给出一个线性时间算法，确定无环无向图（即树）中的最长无权路径。

9.45 考虑  $N \times N$  网格。网格中的一些方格由黑色圆形占据。若两个方格共享一条边，则它们属于同一组。在图9-86中，有一组由4个黑圆占据的方格组成，三组由2个黑圆占据的方格组成，两组由单个黑圆占据的方格组成。假设网格由二维数组表示。编写一个程序进行下列工作：

- 当给出组中的一个方格时计算该组的大小。
- 计算不同的组的个数。
- 列出所有的组。

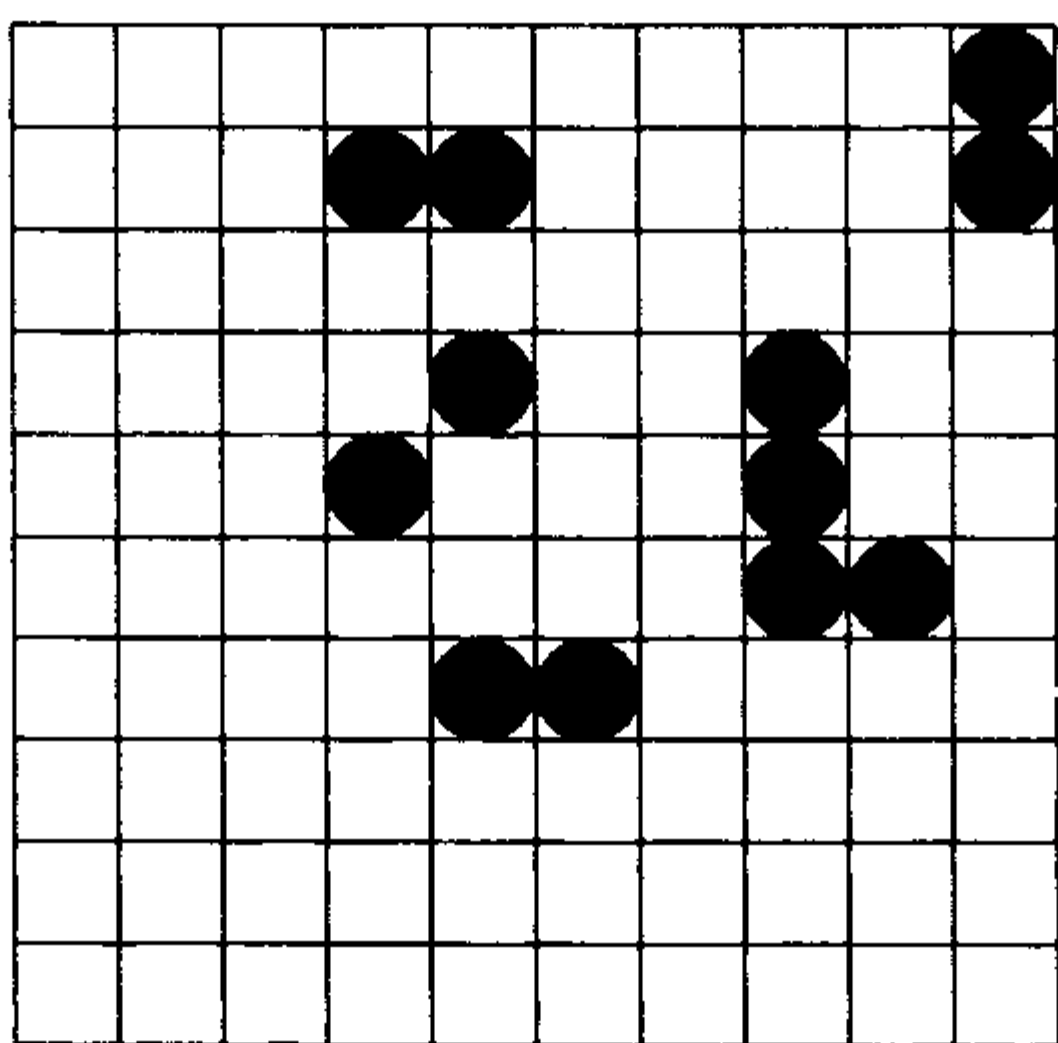


图9-86 练习9.45使用的网格

9.46 8.7节描述了迷宫的生成。设我们想要输出迷宫中的路径。假设迷宫由矩阵表示；矩阵中的每个单元存储关于墙存在（或不存在）的信息。

- 编写一个程序计算输出迷宫中的路径所需要的足够的信息。以SEN...形式（代表向南，然后向东，然后再向北，等等）给出输出结果。
- 如果使用的C++编译器支持图形环境，编写一个程序画出迷宫，且按一个钮则画出路径。

9.47 设迷宫中的墙可以推倒，但要受罚  $P$  个方块。 $P$  为算法指定的参数（如果处罚是0，那么问题很简单）。描述一种算法解决这种类型的问题。你的算法的运行时间是多少？

9.48 设迷宫可以有解也可以没有解。

- 描述一个线性时间算法，该算法确定为了建立一个解而需要推倒的墙的最小面数（提示：用一个双端队列）。
- 描述一个算法（不必是线性的），该算法能够在推倒最小数目的墙之后找到最短路径。注意，a问的解法给不出哪些墙最好被推倒的信息（提示：使用练习9.47）。

9.49 编写一个程序实现字梯游戏。其中，单字母替换的值为1，而单字母的删除或添加的值  $p > 0$  ( $p$  值由使用者指定)。正如9.3.6节末尾提到的，这本质上是一个加权最短路径问题。

解释下列问题（练习9.50~练习9.53）如何应用最短路径算法解出。然后设计一种结构表示输出，并编写一个程序求解相应的问题。

9.50 输入是一组联赛成绩得分（没有平局）。如果所有的队至少有一场赢和一场输，那么通常可以通过愚蠢的传递性论证证明，任一队都比别的队强。例如，在六队联赛中，每队进行3局比赛，设有下列结果： $A$ 胜 $B$ 和 $C$ ； $B$ 胜 $C$ 和 $F$ ； $C$ 胜 $D$ ； $D$ 胜 $E$ ； $E$ 胜 $A$ ； $F$ 胜 $D$ 和 $E$ 。此时可以证明 $A$ 比 $F$ 强，因为 $A$ 胜 $B$ 而 $B$ 又胜了 $F$ 。类似地，还可以证明 $F$ 比 $A$ 强，因为 $F$ 胜 $E$ 而 $E$ 又胜了 $A$ 。给定一组比赛得分和

- 两支运动队 $X$ 和 $Y$ ，或者找出一个证明（若存在的话） $X$ 比 $Y$ 强，或者指出找不到这种形式的证明。
- 9.51 设输入为一组货币和它们的兑换率。是否存在一种兑换顺序能够立刻赚到钱？例如，货币是 $X$ 、 $Y$ 和 $Z$ ，兑换率为 $1X$ 等于 $2Y$ ， $1Y$ 等于 $2Z$ ，而 $1X$ 等于 $3Z$ 。此时， $300Z$ 将买到 $100X$ ，而 $100X$ 又能买到 $200Y$ ，而后者将换到 $400Z$ 。这样，就得到33%的收益。
- 9.52 一名学生需要选修一定量的课程才可获得学位，而课程的选取必须遵守选修顺序。假设每个学期都提供所有的课程，并设学生可以选修任意多门课程。给定课程表和它们的选修顺序，计算出需要最少学期数的课程表。
- 9.53 Kevin Bacon游戏的目标是通过分享的电影角色把电影演员和Kevin Bacon链接起来。链接的最少次数为演员的Bacon数。例如，Tom Hanks的Bacon数为1；他在Apollo 13中与Kevin Bacon分享角色。Sally Field的Bacon数是2，因为她在电影Forrest Gump中与Tom Hanks分享角色，而后者又在电影Apollo 13中与Kevin Bacon分享角色。几乎所有著名演员的Bacon数都是1或者2。假设你有一个广泛的演员表，包含他们所演的角色<sup>1</sup>，完成下列工作：
- 解释如何查找演员的Bacon数。
  - 解释如何查找具有最高Bacon数的演员。
  - 解释如何查找任意两个演员之间的最小链接次数。
- 9.54 团问题可以叙述如下：给定无向图 $G = (V, E)$ 和一个整数 $K$ ， $G$ 包含最少有 $K$ 个顶点的完全子图吗？
- 顶点覆盖问题可以叙述如下：给定无向图 $G = (V, E)$ 和一个整数 $K$ ， $G$ 是否包含一个子集 $V' \subset V$ 使得 $|V'| \leq K$ 并且 $G$ 的每条边都有一个顶点在 $V'$ 中？证明团问题可以多项式地归约成顶点覆盖问题。
- 9.55 设哈密尔顿回路问题对无向图是NP完全的。
- 证明哈密尔顿回路问题对有向图是NP完全的。
  - 证明无权简单最长路径问题对有向图是NP完全的。
- 9.56 棒球卡收藏家问题如下：给定卡片包 $P_1, P_2, \dots, P_M$ 以及一个整数 $K$ ，其中每个包包含年度棒球卡的一个子集，问是否可能通过选择小于等于 $K$ 个包而搜集到所有的棒球卡？证明棒球卡收藏家问题是NP完全的。

403

## 参考文献

好的图论教科书有[8]、[13]、[22]和[37]。更深入的论题，包括对运行时间更为仔细的考虑，见[39]、[41]和[48]。

邻接表的使用是在[24]中倡导的。拓扑排序算法来自[29]，其描述如[34]。Dijkstra算法初现于[9]，应用 $d$ 堆和斐波那契堆的改进分别在[28]和[15]中描述。具有负权的边的最短路径算法归于Bellman[3]；Tarjan[48]描述了保证终止的更为有效的算法。

Ford和Fulkerson关于网络流的开创性工作[14]。沿最短路径增长或在容许最大流增加的路径上增长的想法源自[12]。对该问题的其他一些处理方法可在[10]、[32]、[21]、[6]和[33]中找到。关于最小值流问题的一个算法见于[19]。

早期的最小生成树算法可以在[4]中找到。Prim算法取自[42]，Kruskal算法源自[35]。两个 $O(|E| \log \log |V|)$ 算法是[5]和[49]。理论上一些著名算法出现在[15]、[17]和[30]。这些算法的经验性研究提出，用decreaseKey实现的Prim算法在实践中对于大多数图而言是最好的[40]。

关于双连通性的算法来自[44]。第一个线性时间强分支算法（练习9.28）也出现在这篇论文中。书中出现的算法归于Kosaraju（未发表）和Sharir[43]。深度优先搜索的另外一些应用见于[25]、[26]、[45]和[46]（正如第8章提到的，[45]和[46]中的结果已被改进，但是基本算法没变）。

1. 例如，可参考Internet Movie Database文件actors.list.gz和actress.list.gz，网址为ftp://uiarchive.cs.uiuc.edu/pub/info/imdb/。

NP完全问题理论的经典的介绍性著作是[20], 在[1]中可以找到另外的材料。可满足性的NP完全性在[7]中证明。另一篇开创性的论文是[31], 它证明了21个问题的NP完全性。复杂性理论的一个极好的概括性论述是[47]。旅行商问题的一个近似算法可在[38]中找到, 它给出几近最优的结果。

练习9.8的解法可以在[2]中找到。练习9.13中二分匹配问题的解法见于[23]和[36], 该问题可通过给边加权并除掉图是二分的限制而得以推广。一般图的无权匹配问题的有效解法是相当复杂的, 细节可以在[11]、[16]和[18]中找到。

练习9.35处理平面图, 它通常产生于实践。平面图是非常稀疏的, 许多困难问题以平面图的方式处理会更容易。有一个例子是图的同构问题, 对于平面图它是线性时间可解的[27]。对于一般的图, 尚不知有多项式时间算法。

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. R. K. Ahuja, K. Melhorn, J. B. Orlin, and R. E. Tarjan, "Faster Algorithms for the Shortest Path Problem," *Journal of the ACM*, 37 (1990), 213-223.
3. R. E. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics*, 16 (1958), 87-90.
4. O. Borůvka, "Ojistém problému minimálním(On a Minimal Problem)," *Práce Moravské Přírodovědecké Společnosti*, 3 (1926), 37-58.
5. D. Cheriton and R. E. Tarjan, "Finding Minimum Spanning Trees," *SIAM Journal on Computing*, 5 (1976), 724-742.
6. J. Cheriyan and T. Hagerup, "A Randomized Maximum-Flow Algorithm," *SIAM Journal on Computing*, 24 (1995), 203-226.
7. S. Cook, "The Complexity of Theorem Proving Procedures," *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (1971), 151-158.
8. N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, Englewood Cliffs, N.J., 1974.
9. E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, 1 (1959), 269-271.
10. E. A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation," *Soviet Mathematics Doklady*, 11 (1970), 1277-1280.
11. J. Edmonds, "Paths, Trees, and Flowers," *Canadian Journal of Mathematics*, 17 (1965), 449-467.
12. J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *Journal of the ACM*, 19 (1972), 248-264.
13. S. Even, *Graph Algorithms*, Computer Science Press, Potomac, Md., 1979.
14. L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, N.J., 1962.
15. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM*, 34 (1987), 596-615.
16. H. N. Gabow, "Data Structures for Weighted Matching and Nearest Common Ancestors with Linking," *Proceedings of First Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), 434-443.
17. H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, "Efficient Algorithms for Finding Minimum Spanning Trees on Directed and Undirected Graphs," *Combinatorica*, 6 (1986), 109-122.
18. Z. Galil, "Efficient Algorithms for Finding Maximum Matchings in Graphs," *ACM Computing Surveys*, 18 (1986), 23-38.
19. Z. Galil and E. Tardos, "An  $O(n^2(m+n \log n) \log n)$  Min-Cost Flow Algorithm," *Journal of the ACM*, 35 (1988), 374-386.
20. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.
21. A. V. Goldberg and R. E. Tarjan, "A New Approach to the Maximum-Flow Problem," *Journal of the ACM*, 35 (1988), 921-940.
22. F. Harary, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.
23. J. E. Hopcroft and R. M. Karp, "An  $n^{5/2}$  Algorithm for Maximum Matchings in Bipartite Graphs," *SIAM*



- Journal on Computing*, 2 (1973), 225-231.
24. J. E. Hopcroft and R. E. Tarjan, "Algorithm 447: Efficient Algorithms for Graph Manipulation," *Communications of the ACM*, 16 (1973), 372-378.
  25. J. E. Hopcroft and R. E. Tarjan, "Dividing a Graph into Triconnected Components," *SIAM Journal on Computing*, 2 (1973), 135-158.
  26. J. E. Hopcroft and R. E. Tarjan, "Efficient Planarity Testing," *Journal of the ACM*, 21 (1974), 549-568.
  27. J. E. Hopcroft and J. K. Wong, "Linear Time Algorithm for Isomorphism of Planar Graphs," *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing* (1974), 172-184.
  28. D. B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM*, 24 (1977), 1-13.
  29. A. B. Kahn, "Topological Sorting of Large Networks," *Communications of the ACM*, 5 (1962), 558-562.
  30. D. R. Karger, P. N. Klein, and R. E. Tarjan, "A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees," *Journal of the ACM*, 42 (1995), 321-328.
  31. R. M. Karp, "Reducibility among Combinatorial Problems," *Complexity of Computer Computations* (eds. R. E. Miller and J. W. Thatcher), Plenum Press, New York, 1972, 85-103.
  32. A. V. Karzanov, "Determining the Maximal Flow in a Network by the Method of Preflows," *Soviet Mathematics Doklady*, 15 (1974), 434-437.
  33. V. King, S. Rao, and R. E. Tarjan, "A Faster Deterministic Maximum Flow Algorithm," *Journal of Algorithms*, 17 (1994), 447-474.
  34. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd. ed., Addison-Wesley, Reading, Mass., 1997.
  35. J. B. Kruskal, Jr., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society*, 7 (1956), 48-50.
  36. H. W. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly*, 2 (1955), 83-97.
  37. E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Reinhart, and Winston, New York, 1976.
  38. S. Lin and B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem," *Operations Research*, 21 (1973), 498-516.
  39. K. Melhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-completeness*, Springer-Verlag, Berlin, 1984.
  40. B. M. E. Moret and H. D. Shapiro, "An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree," *Proceedings of the Second Workshop on Algorithms and Data Structures* (1991), 400-411.
  41. C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, Englewood Cliffs, N.J., 1982.
  42. R. C. Prim, "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal*, 36 (1957), 1389-1401.
  43. M. Sharir, "A Strong-Connectivity Algorithm and Its Application in Data Flow Analysis," *Computers and Mathematics with Applications*, 7 (1981), 67-72.
  44. R. E. Tarjan, "Depth First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, 1 (1972), 146-160.
  45. R. E. Tarjan, "Testing Flow Graph Reducibility," *Journal of Computer and System Sciences*, 9 (1974), 355-365.
  46. R. E. Tarjan, "Finding Dominators in Directed Graphs," *SIAM Journal on Computing*, 3 (1974), 62-89.
  47. R. E. Tarjan, "Complexity of Combinatorial Algorithms," *SIAM Review*, 20 (1978), 457-491.
  48. R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 1983.
  49. A. C. Yao, "An  $O(|E| \log \log |V|)$  Algorithm for Finding Minimum Spanning Trees," *Information Processing Letters*, 4 (1975), 21-23.



至此，我们已经涉及一些算法的有效实现。可以看出，当给定一个算法时，实际的数据结构无需指定。为使运行时间尽可能短，需要由编程人员来选择适当的数据结构。

本章将把注意力从算法的实现转向算法的设计。到现在为止我们已经看到的大部分算法都是直接且简单的。第9章包含的一些算法要深奥得多，有些需要（在有些情形下很长的）论据证明它们确实是正确的。本章将集中讨论用于求解问题的5种常见类型的算法。对于许多问题，很可能这些方法中至少有一种方法是可以解决问题的。对于每种类型的算法，我们将：

- 了解一般的处理方法。
- 考察几个例子（本章末尾的习题提供了更多的例子）。
- 在适当的地方概括地讨论时间和空间复杂性。

### 10.1 贪心算法

我们将要考察的第一种类型的算法是贪心算法（greedy algorithm）。在第9章我们已经看到三个贪心算法：Dijkstra算法、Prim算法和Kruskal算法。贪心算法分阶段工作。在每一个阶段，可以认为所做的决定是好的，而不考虑将来的后果。一般来说，这意味着选择的是某个局部最优。这种“眼下能够拿到的就拿”的策略即是这类算法名称的来源。当算法终止时，我们希望局部最优就是全局最优。如果真是这样的话，那么算法就是正确的；否则，算法得到的是一个次最优解。如果不要绝对的最佳答案，那么有时用简单的贪心算法生成近似答案，而不是使用一般来说产生准确答案所需要的复杂算法。

409

有几个现实的贪心算法的例子。最明显的是找零钱问题。为了使用美国货币找零钱，我们重复地配发最大面值货币。于是，为了找出17美元61美分的零钱，我们拿出一张10元美钞、一张5元美钞、两张1元美钞、两个25分币、一个10分币以及一个1分币。这么做可以保证使用最少的钞票和硬币。这个算法不是对所有的货币系统都行得通，但幸运的是，可以证明它对美国货币系统是正确的。事实上，即使允许使用2元美钞和50美分币该算法也仍然是可行的。

交通问题有一个例子，在这个例子中，进行局部最优选择并不总是行得通的。例如，在迈阿密的某些交通高峰期间，即使一些主要马路看起来空荡荡的，也最好还是把车停在这些街道以外，因为车流将会沿着马路阻塞一英里长，你也就被堵在那里动弹不得。有时甚至更糟，为了回避所有的交通瓶颈，最好是朝着与你的目的地相反的方向临时绕道行驶。

本节其余部分将考察几个使用贪心算法的应用。第一个应用是简单的调度问题。实际上，所有的调度问题或者是NP完全的（或类似的难度），或者是贪心算法可解的。第二个应用处理文件压缩，这是计算机科学最早的成果之一。最后，将介绍一个贪心近似算法的例子。

### 10.1.1 一个简单的调度问题

现有作业 $j_1, j_2, \dots, j_N$ ，已知对应的运行时间分别为 $t_1, t_2, \dots, t_N$ ，而处理器只有一个。为了把作业平均完成的时间最小化，调度这些作业最好的方式是什么？本节将假设非抢占调度(nonpreemptive scheduling)：一旦开始一个作业，就必须把该作业运行完。

作为一个例子，设四个作业和相关的运行时间如图10-1所示。一个可能的调度在图10-2中给出。因为 $j_1$ 用15个时间单位， $j_2$ 到23个时间单位， $j_3$ 到26个时间单位，而 $j_4$ 到36个时间单位，所以平均完成时间为25。一个更好的调度如图10-3所示，它产生的平均完成时间为17.75。

| 作业    | 时间 |
|-------|----|
| $j_1$ | 15 |
| $j_2$ | 8  |
| $j_3$ | 3  |
| $j_4$ | 10 |

图10-1 作业和时间

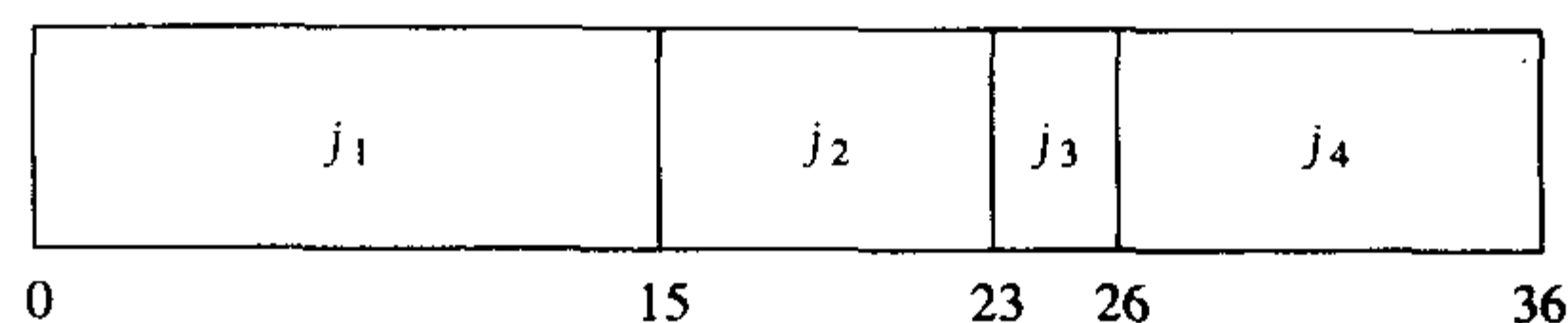


图10-2 1号调度

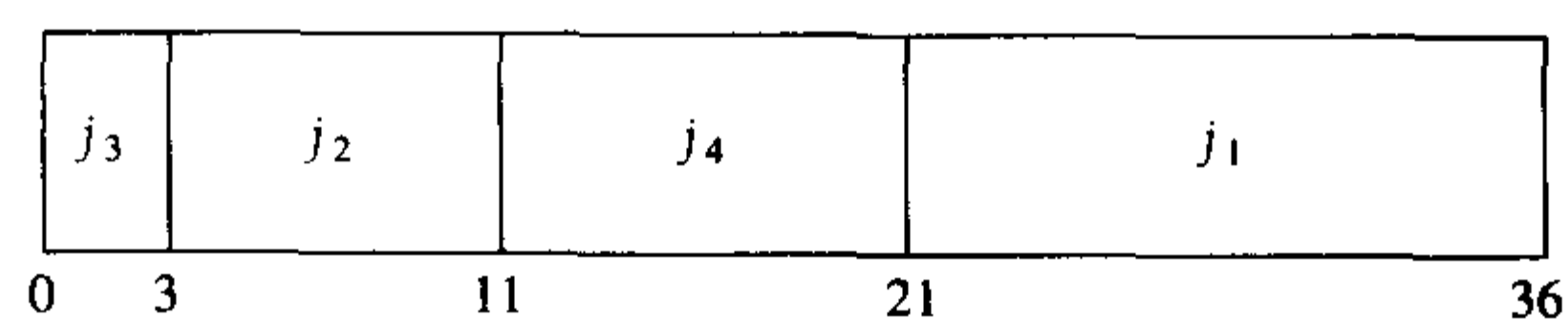


图10-3 2号调度（最优）

图10-3给出的调度是按照最短的作业最先进行来安排的。可以证明这将总会产生一个最优的调度。令调度表中的作业是 $j_{i_1}, j_{i_2}, \dots, j_{i_N}$ 。第一个作业以时间 $t_{i_1}$ 完成，第二个作业在 $t_{i_1} + t_{i_2}$ 后完成，而第三个作业在 $t_{i_1} + t_{i_2} + t_{i_3}$ 后完成。由此可以看到，该调度的总值 $C$ 为：

$$C = \sum_{k=1}^N (N - k + 1) t_{i_k} \quad (10-1)$$

$$C = (N + 1) \sum_{k=1}^N t_{i_k} - \sum_{k=1}^N k \cdot t_{i_k} \quad (10-2)$$

注意，在式(10-2)中第一个和与作业的排序无关，因此只有第二个和影响到总值。设在一个排序中存在某个 $x > y$ 使得 $t_{i_x} < t_{i_y}$ 。此时，计算表明，交换 $j_{i_x}$ 和 $j_{i_y}$ ，第二个和增加，从而降低了总值。因此，所用时间不是单调非减的任何作业的调度必然是次最优的。剩下的只有那些其作业按照最小运行时间最先安排的调度才是所有调度方案中最优的。

这个结果指出了操作系统调度程序一般把优先权赋予那些更短的作业的原因。

#### 1. 多处理器的情形

可以把这个问题扩展到多个处理器的情形。设有作业 $j_1, j_2, \dots, j_N$ ，对应的运行时间分别为 $t_1, t_2, \dots, t_N$ ，另有处理器的个数 $P$ 。不失一般性，假设作业是有序的，最短的最先运行。作为一个例子，设 $P = 3$ ，而作业则如图10-4所示。

图 10-5 显示了一个最优的安排，它把平均完成时间优化到最小。作业  $j_1$ 、 $j_4$  和  $j_7$  在处理器 1 上运行。处理器 2 处理作业  $j_2$ 、 $j_5$  和  $j_8$ ，而处理器 3 运行其余的作业。总的完成时间为 165，平均是  $\frac{165}{9} = 18.33$ 。

| 作业    | 时间 |
|-------|----|
| $j_1$ | 3  |
| $j_2$ | 5  |
| $j_3$ | 6  |
| $j_4$ | 10 |
| $j_5$ | 11 |
| $j_6$ | 14 |
| $j_7$ | 15 |
| $j_8$ | 18 |
| $j_9$ | 20 |

图10-4 作业和时间

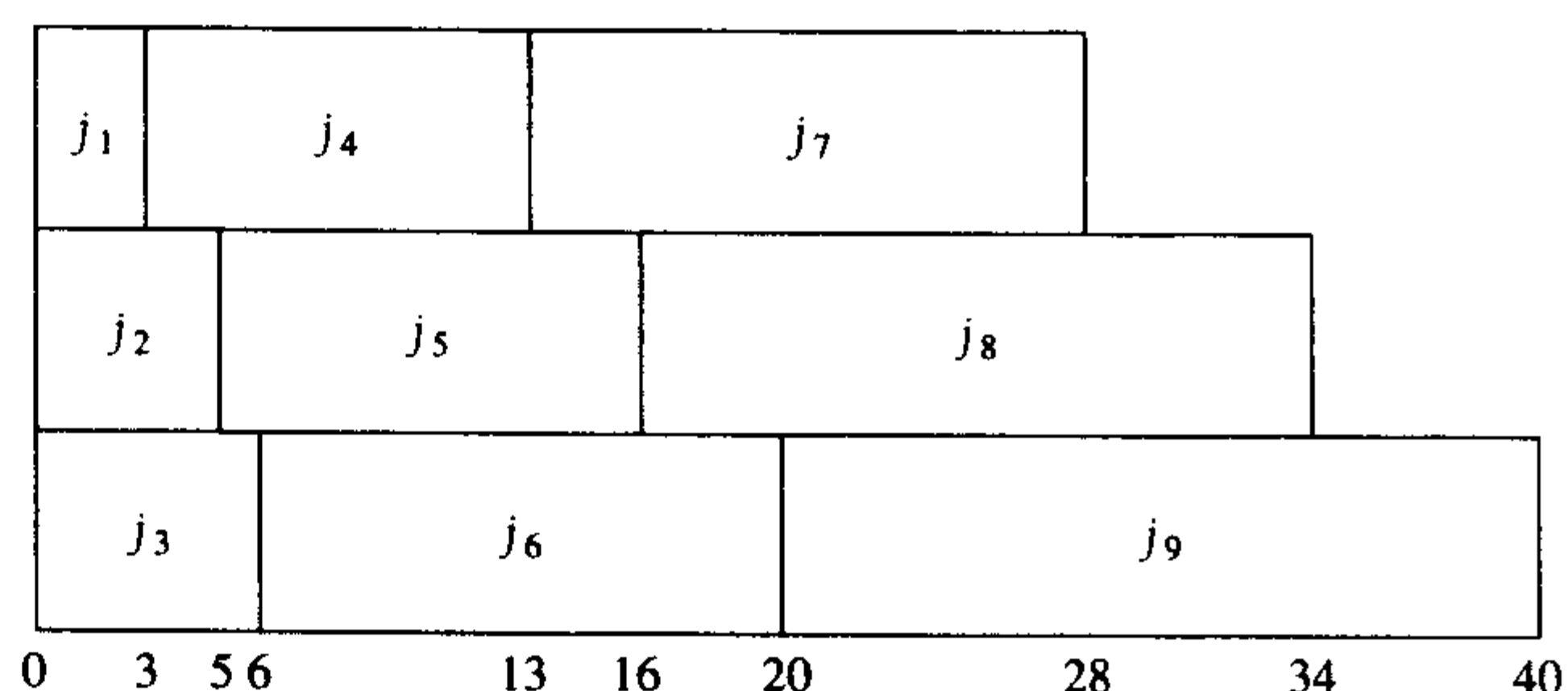


图10-5 多处理器情形的一个最优解

解决多处理器情形的算法是按顺序开始作业的，处理器之间轮换分配作业。不难证明没有其他的顺序能够做得更好，虽然处理器个数  $P$  能够整除作业数  $N$  时存在许多最优的顺序。对于每一个  $0 \leq i < N/P$ ，把从  $j_{iP+1}$  直到  $j_{(i+1)P}$  的每一个作业放到不同的处理器上，可以得到这样的最优顺序。在我们的例子中，图 10-6 显示了第二个最优解。

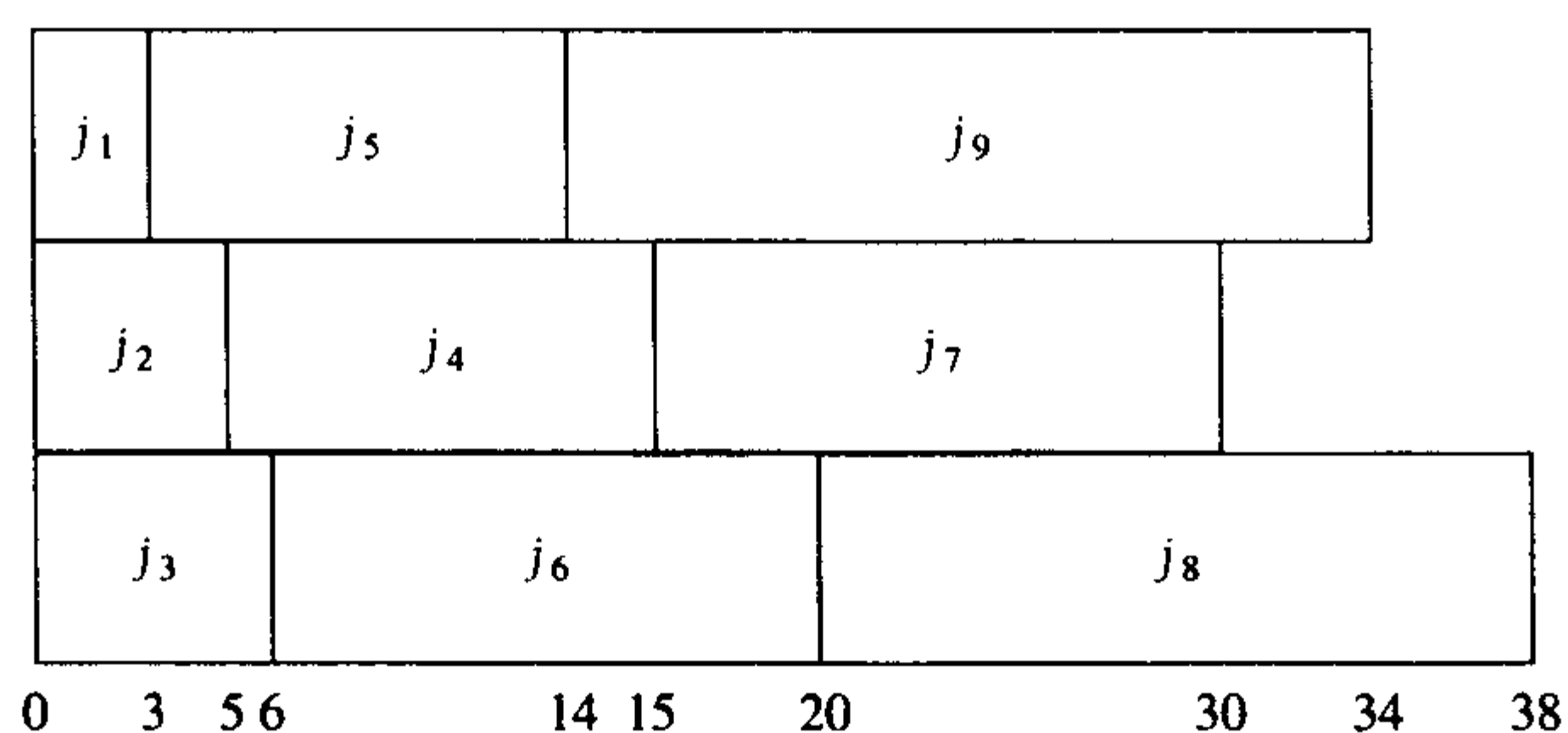


图10-6 多处理器情形的第二个最优解

即使  $P$  不恰好整除  $N$ ，哪怕所有的作业时间是互异的，也还是有许多最优解。我们把进一步的考察留作练习。

## 2. 将最后完成时间最小化

在本节最后，考虑一个非常类似的问题。假设我们只关注最后的作业的完成时间。在上面的两个例子中，它们的完成时间分别是 40 和 38。图 10-7 指出最小的最后完成时间是 34，而这个结果显然不能再改进了，因为每一个处理器都一直在忙着。

虽然这个调度未获得最小平均完成时间，但是它有个优点，即整个序列的完成时间更早。如果同一个用户拥有所有这些作业，那么该调度方法是更可取的。虽然这些问题非常相似，但是这个新问题实际上是 NP 完全的；它恰是背包问题或装箱问题的另一种表述方式，我们在本节后面还将遇到它。因此，将最后完成时间最小化显然要比把平均完成时间最小化困难得多。

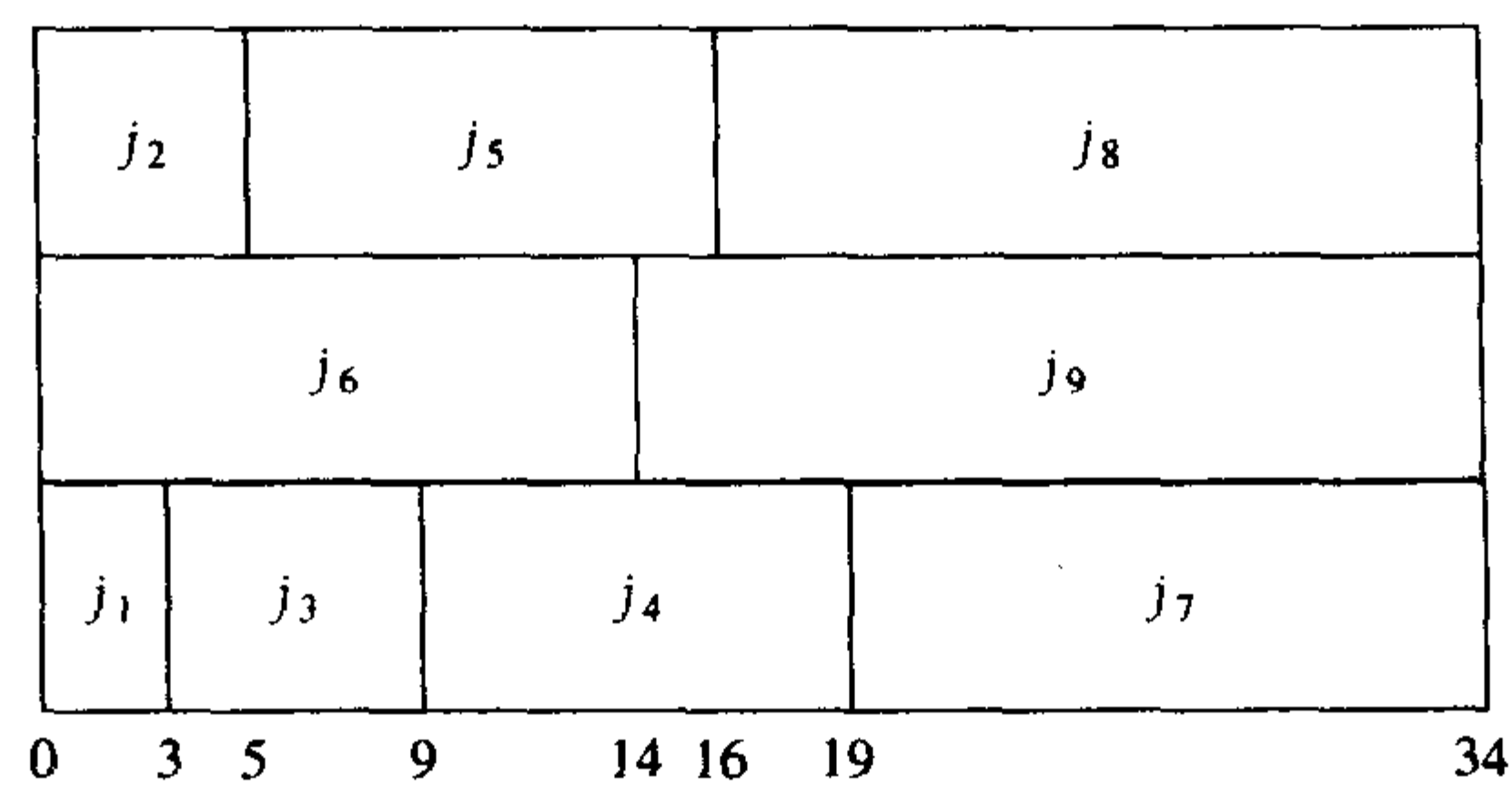


图10-7 将最后完成时间最小化

10.1.2 赫夫曼编码

本节考虑贪心算法的第二个应用，称为文件压缩（file compression）。

413

标准的ASCII字符集由大约100个“可打印”字符组成。为了把这些字符区分开来，需要 $\lceil \log 100 \rceil = 7$ 个位（bit）。但7个位可以表示128个字符，因此ASCII字符还可以再加上一些其他的“不可打印”字符。我们加上第8个位作为奇偶校验位。不过，重要的问题是，如果字符集的大小是C，那么在标准的编码中就需要 $\lceil \log C \rceil$ 个位。

设有一个文件，它只包含字符a、e、i、s、t，加上一些空格和换行（newline）。进一步设该文件有10个a、15个e、12个i、3个s、4个t、13个空格以及1个换行。如图10-8所示，这个文件需要174个位来表示，因为有58个字符而每个字符需要3个位。

| 字符 | 编码  | 频率 | 总位数 |
|----|-----|----|-----|
| a  | 000 | 10 | 30  |
| e  | 001 | 15 | 45  |
| i  | 010 | 12 | 36  |
| s  | 011 | 3  | 9   |
| t  | 100 | 4  | 12  |
| 空格 | 101 | 13 | 39  |
| 换行 | 110 | 1  | 3   |
| 总计 |     |    | 174 |

图10-8 使用一个标准编码方案

在现实中，文件可能相当大。许多非常大的文件是某个程序的输出数据，而在使用频率最高和最低的字符之间通常存在很大的差别。例如，许多大的文件都含有大量的数字、空格和换行，但是q和x却很少。如果在慢速的电话线上传输这些信息，那么就会希望减少文件的大小。还有，由于实际上每台机器上的磁盘空间都是非常珍贵的，因此人们就想知道是否有可能提供一种更好的编码降低所需的总比特数。

答案是肯定的，一种简单的策略可以使典型的大型文件节省25%，而使许多大型的数据文件节省多达50%~60%。这种一般的策略就是让编码的长度随字符变化，同时保证经常出现的字符的编码要短。注意，如果所有的字符都以相同的频率出现，那么要节省空间是不可能的。

代表字母的二进制代码可以用二叉树来表示，如图10-9所示。



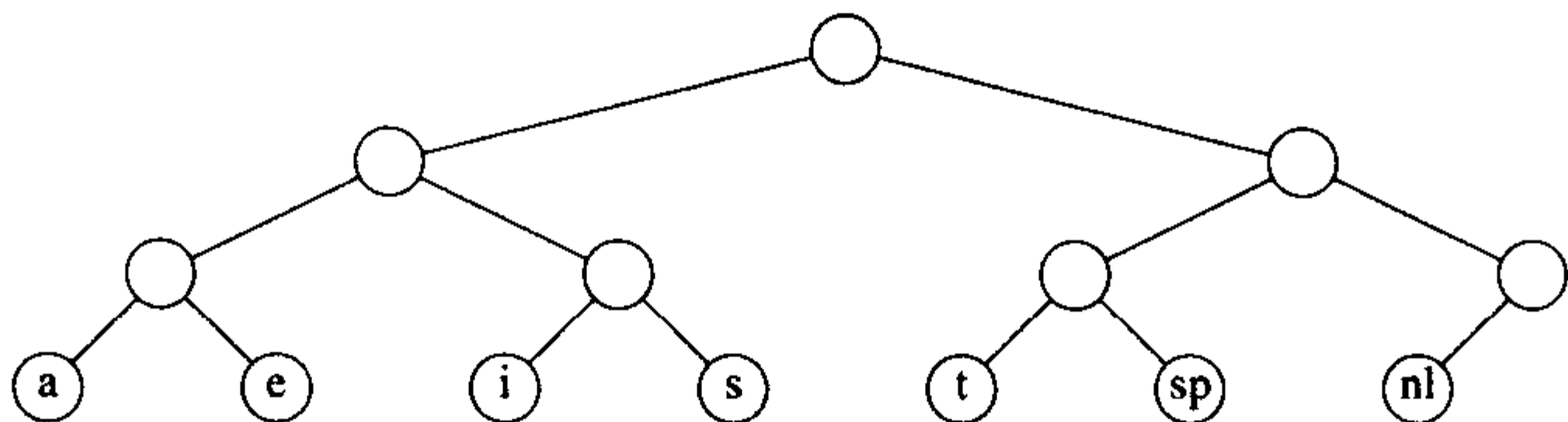


图10-9 树中原始编码的表示

图10-9中的树只在树叶上有数据。每个字符通过从根结点开始，用0指示左分支，用1指示右分支来记录路径的方法表示出来。例如，s通过从根向左走，然后向右，最后再向右而达到，于是它被编码成011。这种数据结构有时称作检索树（trie）。如果字符 $c_i$ 在深度 $d_i$ 处并且出现 $f_i$ 次，那么这种字符编码的值就是 $\sum d_i f_i$ 。

414

可以利用换行唯一是一个儿子而得到一种比图10-9给出的编码更好的编码。通过把换行符号放到其更高一层的父结点上，可以得到图10-10中的新的树。这棵新树的值是173，但该值仍然远没有达到最优。

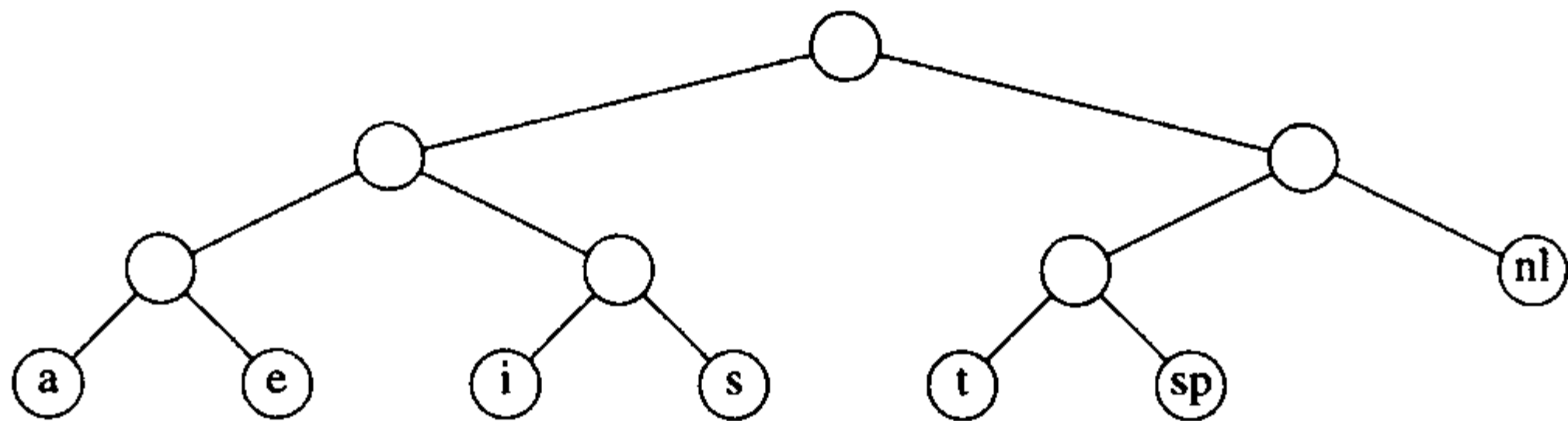


图10-10 稍微好一些的树

注意，图10-10中的树是一棵满树（full tree）：所有的结点要么是树叶，要么有两个儿子。最优的编码将总具有这个性质，否则正如我们已经看到的，具有一个儿子的结点可以向上移动一层。

如果字符都只放在树叶上，那么任何位序列都总能够被毫无歧义地译码。例如，编码串是0100111100010110001000111。0不是字符编码，01也不是字符编码，但010代表i，于是第一个字符是i；然后跟着的是011，它是字符s；其后的11是换行。剩下的编码分别是a、空格、t、i、e和换行。因此，这些字符编码的长度是否不同并不要紧，要紧的是只要没有任何字符编码是别的字符编码的前缀就行。这样的编码称为前缀码（prefix code）。相反，如果一个字符放在非叶结点上，那就不能够再保证译码没有二义性。

综上所述，可以看到，基本的问题在于找到（如上定义的）总值最小的满二叉树，其中所有的字符都位于树叶上。图10-11中的树显示了本例样本字母表的最优树。从图10-12可以看到，这种编码只用了146个位。

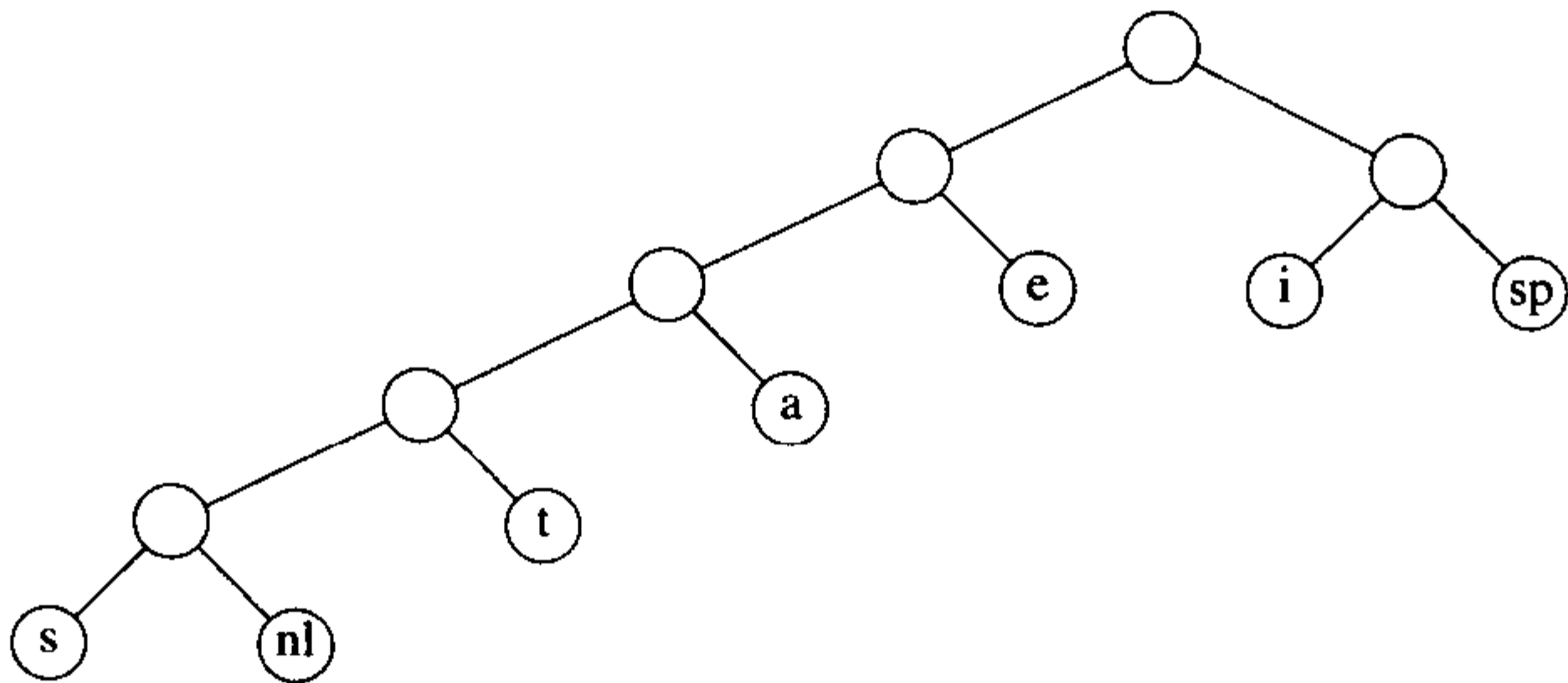


图10-11 最优前缀码的树

| 字符 | 编码    | 频率 | 总位数 |
|----|-------|----|-----|
| d  | 001   | 10 | 30  |
| e  | 01    | 15 | 30  |
| i  | 10    | 12 | 24  |
| s  | 00000 | 3  | 15  |
| t  | 0001  | 4  | 16  |
| 空格 | 11    | 13 | 26  |
| 换行 | 00001 | 1  | 5   |
| 总计 |       |    | 146 |

图10-12 最优前缀码

注意，存在许多的最优编码。这些编码可以通过交换编码树中的儿子结点得到。此时，未解决的主要问题是如何构造编码树。1952年，赫夫曼给出了一个算法，因此，这种编码系统通常称为赫夫曼编码（Huffman code）。

415

赫夫曼算法

本节将假设字符的个数为C。赫夫曼算法（Huffman algorithm）可以描述如下：维护一个由树组成的森林。一棵树的权等于它的叶子的频率的和。任意选取最小权的两棵树T1和T2，并任意形成以T1和T2为子树的新树，将这样的过程进行C-1次。在算法的开始，存在C棵单结点树——每个字符一棵。在算法结束时得到一棵树，这棵树就是最优赫夫曼编码树。

下面通过一个具体例子来说明此算法的操作。图10-13表示的是初始的森林；每棵树的权在根处以小号数字标出。将两棵权最低的树合并到一起，由此创建了图10-14中的森林。将新的根命名为T1，这样可以无二义性地表述进一步的合并。图10-14中令s是左儿子，这里，令其为左儿子还是右儿子是任意的；注意可以使用赫夫曼算法描述中的两个任意性。新树的总的权正是那些老树的权的和，当然也就很容易计算。由于创建新树只需得出一个新结点，设置左指针和右指针并把权记录下来，因此创建新树很简单。

416

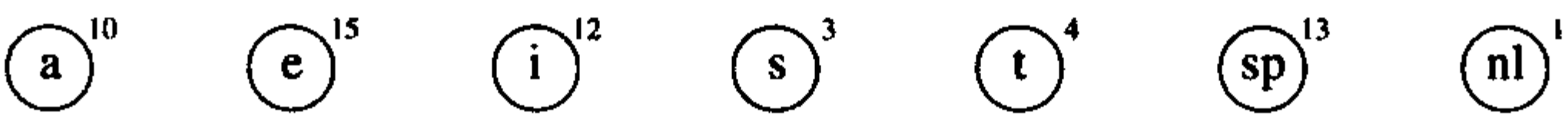


图10-13 赫夫曼算法的初始状态

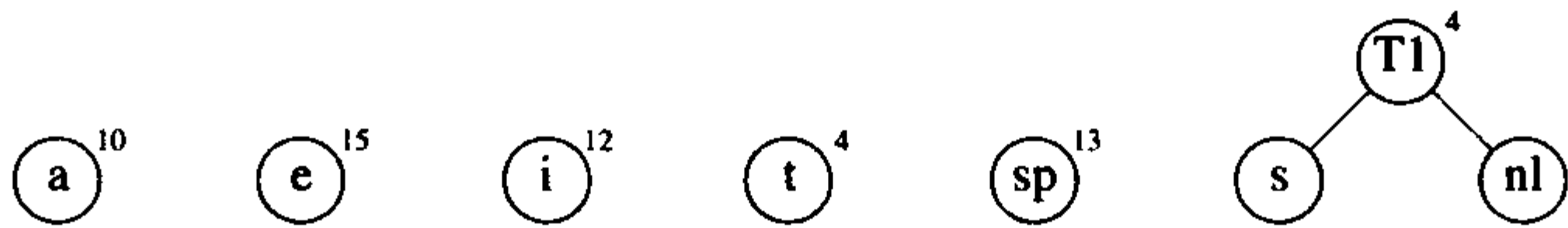


图10-14 第一次合并后的赫夫曼算法

现在有六棵树，再选取两棵权最小的树。这两棵树是T1和t，然后将它们合并成一棵新树，树根在T2，权是8，见图10-15。第三步将T2和a合并建立T3，其权为10 + 8 = 18。图10-16显示了这次操作的结果。



图10-15 第二次合并后的赫夫曼算法

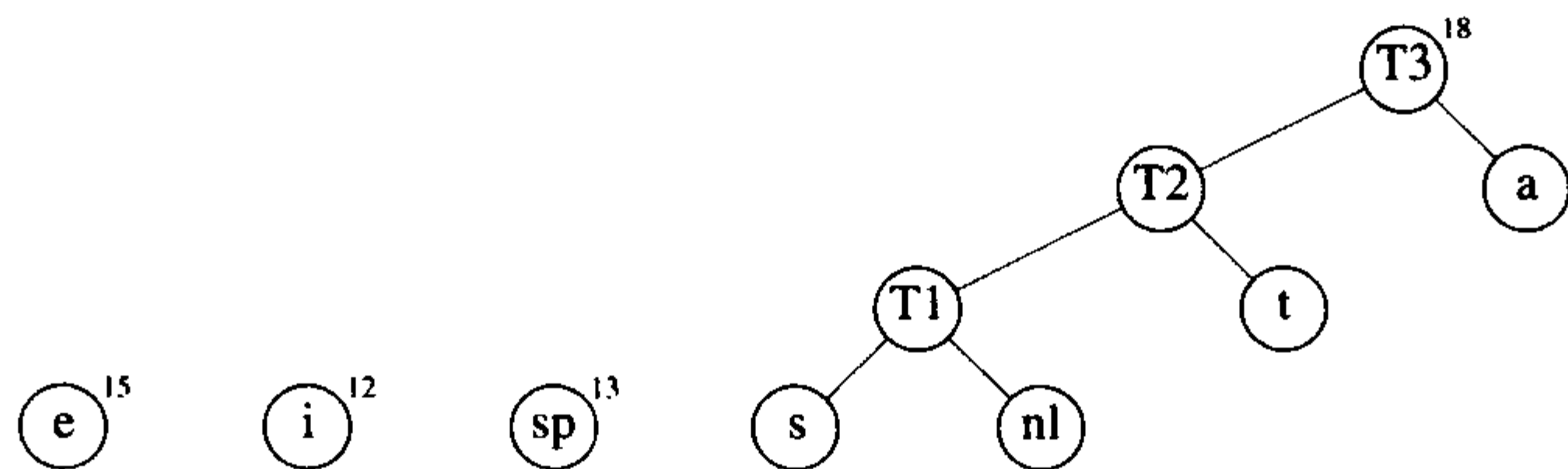


图10-16 第三次合并后的赫夫曼算法

在第三次合并完成后，最低权的两棵树是代表*i*和空格的两个单结点树。图10-17指出这两棵树如何合并成根为T4的新树。第五步合并根为*e*和T3的树，因为这两棵树的权最小。该步结果如图10-18所示。

417

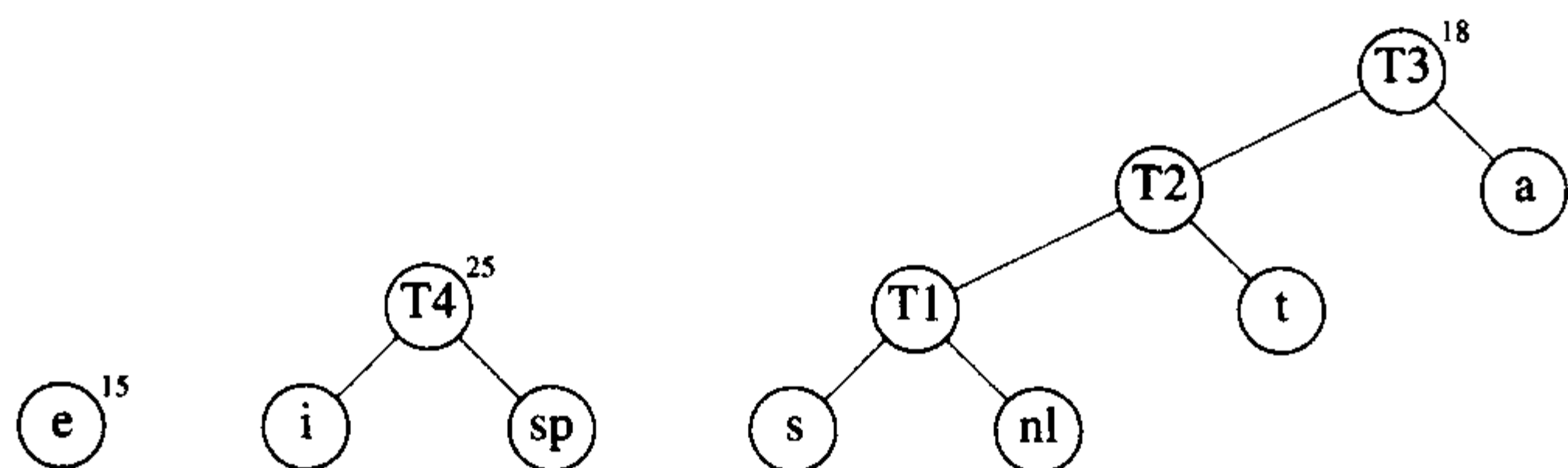


图10-17 第四次合并后的赫夫曼算法

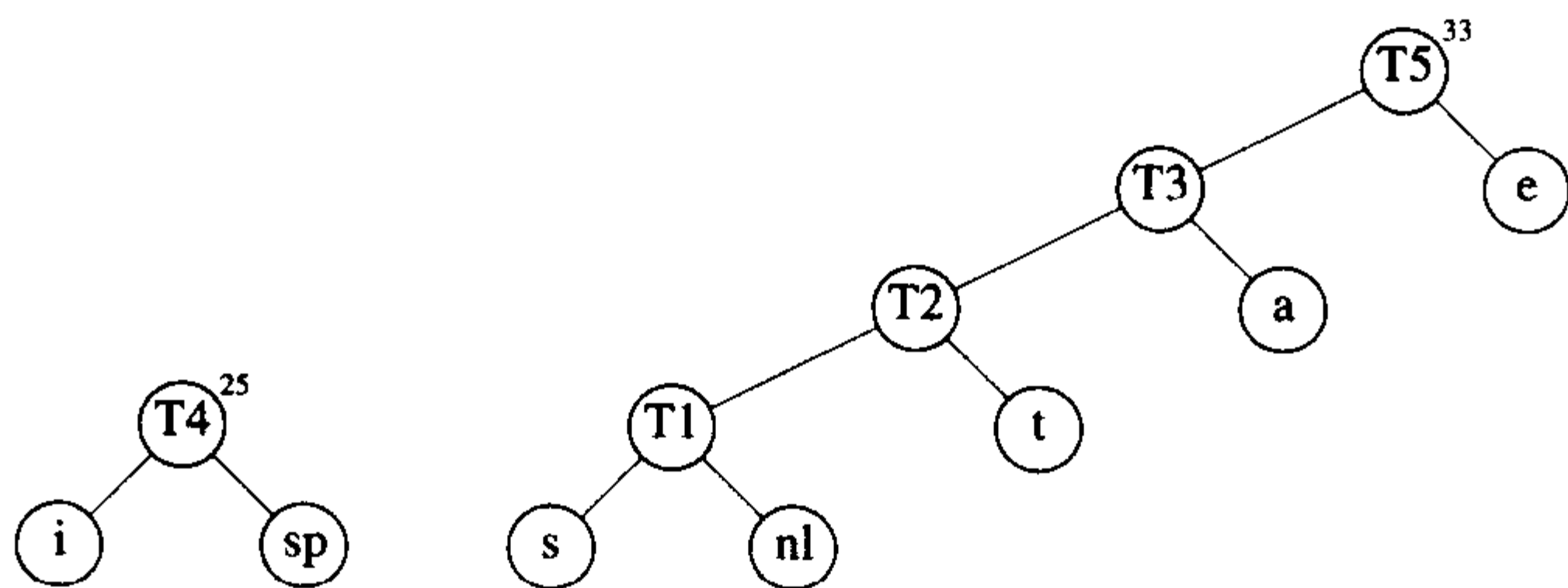


图10-18 第五次合并后的赫夫曼算法

最后，将两棵剩下的树合并，得到图10-11所示的最优树。图10-19显示出这棵最优树，其根为T6。

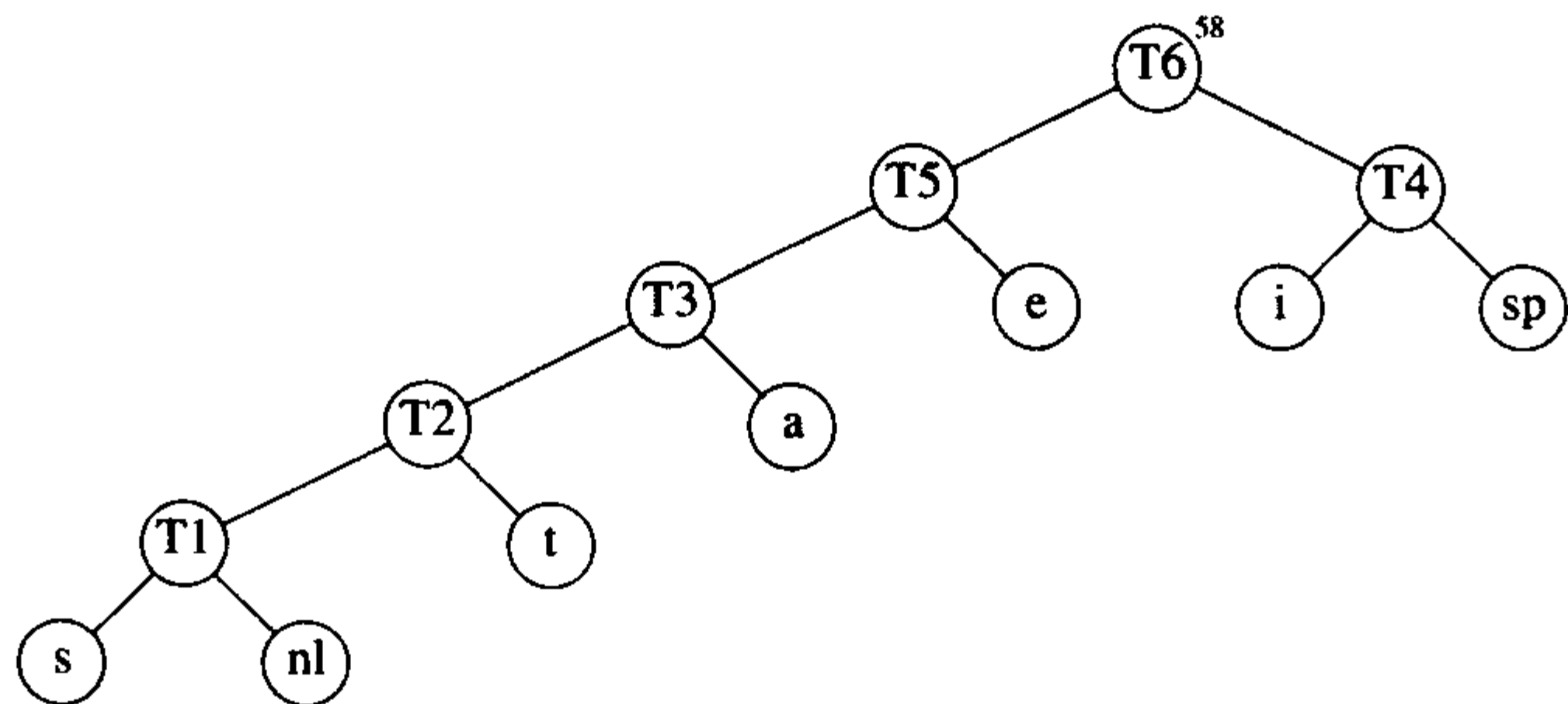


图10-19 最后一次合并后的赫夫曼算法

我们将概述赫夫曼算法产生最优代码的证明思路，详细的细节将留作练习。首先，由反证法不难证明树必然是满的，因为我们已经看到一棵不满的树是如何改进成满树的。

其次，必须证明两个出现频率最小的字符 $\alpha$ 和 $\beta$ 必是两个最深的结点（虽然其他结点可以同

样地深)。这通过反证法同样容易证明, 因为, 如果 $\alpha$ 或 $\beta$ 不是最深的结点, 那么必然存在某个 $\gamma$ 是最深的结点(记住树是满的)。如果 $\alpha$ 的频率小于 $\gamma$ , 那么可以通过交换它们在树中的位置而改进权值。

然后可以论证, 在相同深度上任意两个结点处的字符可以交换而不影响最优性。这说明, 总可以找到一棵最优树, 它含有两个最不经常出现的符号作为兄弟; 因此第一步没有错。

证明可以通过归纳法论证来完成。当树被合并时, 我们认为新的字符集是在根上的那些字符。于是, 在我们的例子中, 经过四次合并以后, 可以把字符集看成由e与元字符T3和T4组成。这恐怕是证明最微妙的部分; 我们要求读者补足所有的细节。

该算法是贪心算法的原因在于, 在每一阶段都进行一次合并而没有进行全局的考虑, 只是选择两棵最小的树。

418

如果依权排序将这些树保存在一个优先队列中, 那么, 对元素个数不会超过 $C$ 的优先队列将进行一次buildHeap、 $2C-2$ 次deleteMin和 $C-2$ 次insert, 因此运行时间为 $O(C \log C)$ 。若使用一个链表简单实现该优先队列, 则将给出一个 $O(C^2)$ 算法。优先队列实现方法的选择取决于 $C$ 有多大。在ASCII字符集的情况下,  $C$ 是足够小的, 这使得二次的运行时间是可以接受的。在这样的应用中, 实际上所有的运行时间都将花费在读进输入文件和写出压缩文件所需要的磁盘I/O上。

有两个细节必须要考虑。首先, 在压缩文件的开头必须要传送编码信息, 否则将不可能译码。做这件事有几种方法, 见练习10.4。对于一些小文件, 传送编码信息表的代价将超过压缩带来的任何可能的节省, 最后的结果很可能是文件扩大。当然, 这可以检测到, 并且原始文件可原样保留。对于大型文件, 信息表的大小是无关紧要的。

第二个问题, 该算法是一个两趟扫描算法。第一趟搜集频率数据, 第二趟进行编码。显然, 对于处理大型文件的程序来说这个性质不是我们所希望的。某些替代的做法在参考文献中进行了介绍。

### 10.1.3 近似装箱问题

本节将考虑某些解决装箱问题(bin packing problem)的算法。这些算法运行得很快, 但未必产生最优解。不过, 我们将证明所产生的解接近最优解。

设给定 $N$ 项物品, 大小为 $s_1, s_2, \dots, s_N$ , 所有的大小都满足 $0 < s_i \leq 1$ 。问题是要把这些物品装到最少数量的箱子中去, 已知每个箱子的容量是1个单位。作为例子, 图10-20显示了把大小为0.2、0.5、0.4、0.7、0.1、0.3、0.8的一批物品最优装箱的方法。

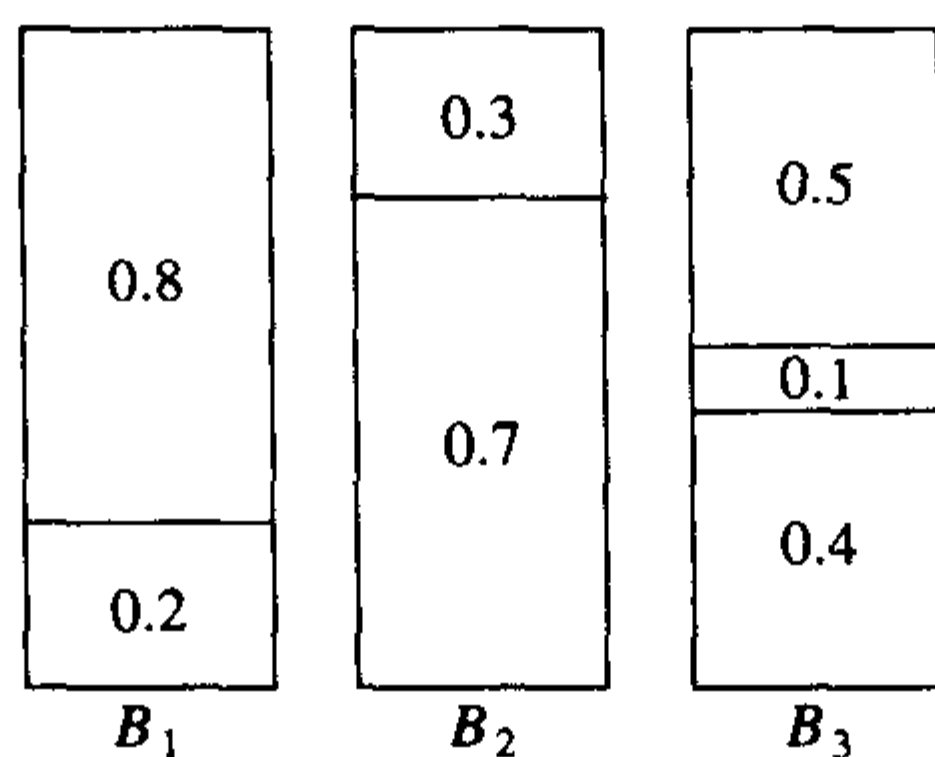


图10-20 对0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8的最优装箱

有两种版本的装箱问题。第一种是联机装箱(on-line packing)问题。在这种问题中, 每一件物品必须放入一个箱子之后才处理下一件物品。第二种是脱机装箱(off-line packing)问题。在脱机装箱算法中, 做任何事都需要等到所有的输入数据全被读入之后才进行。联机算法和脱机

419



算法之间的区别在8.2节讨论过。

### 1. 联机算法

要考虑的第一个问题是，联机算法是否实际上总能给出最优的解答，即使在允许无限计算的情况下。我们知道，即使允许无限计算，联机算法也必须先放入一件物品然后才能处理下一件物品，并且不能改变决定。

为了证明联机算法不能总给出最优解，我们将给它一组特别难的数据来处理。考虑由重量为  $\frac{1}{2} - \varepsilon$  的  $M$  个小项后跟重量为  $\frac{1}{2} + \varepsilon$  的  $M$  个大项构成的序列  $I_1$ ，其中  $0 < \varepsilon < 0.01$ 。显然，如果我们在每个箱子中放一个小项再放一个大项，那么这些项可以放入到  $M$  个箱子中。假设存在一个最优联机算法  $A$  可以进行这项装箱工作。考虑算法  $A$  对序列  $I_2$  的操作，该序列只由重量为  $\frac{1}{2} - \varepsilon$  的  $M$  个小项组成。 $I_2$  是可以装入  $\lceil M/2 \rceil$  个箱子中的。然而，由于  $A$  对序列  $I_2$  的处理结果必然和对  $I_1$  的前半部分的处理结果相同，而  $I_1$  前半部分的输入跟  $I_2$  的输入完全相同，因此  $A$  将把每一项放到一个单独的箱子内。这说明  $A$  将使用的箱子数是序列  $I_2$  最优解的两倍。这样证明了，对于联机装箱问题不存在最优算法。

上面的论述指出，联机算法从不知道输入何时会结束，因此它提供的任何性能保证必须在整个算法的每一时刻都成立。如果我们遵循前面的策略，那么可以证明下列定理。

**定理10.1** 存在一些输入使得任意联机装箱算法至少使用最优箱子数的  $\frac{4}{3}$ 。

**证明** 假设情况相反，为简单起见设  $M$  是偶数。考虑任一运行在上面的输入序列  $I_1$  上的联机算法  $A$ 。注意，该序列由  $M$  个小项后接  $M$  个大项组成。让我们考虑该算法在处理第  $M$  项后都做了什么。设  $A$  已经用了  $b$  个箱子。此时，箱子的最优个数是  $M/2$ ，因为可以在每个箱子里放入两件物品。

根据好于  $\frac{4}{3}$  的性能保证的假设，得知  $2b/M < \frac{4}{3}$ 。

现在考虑在所有的物品都被装箱后算法  $A$  的性能。在第  $b$  个箱子之后开辟的所有箱子中每箱恰好包含一项物品，因为所有小物品都被放在了前  $b$  个箱子中，而两个大项物品又装不进一个箱子中。由于前  $b$  个箱子每箱最多有两项物品，而其余的箱子每箱都有一项物品，因此我们看到，将  $2M$  项物品装箱将至少需要  $2M - b$  个箱子。但  $2M$  项物品可以用  $M$  个箱子最优装箱，因此性能保证可以确保得到  $(2M - b)/M < \frac{4}{3}$ 。

第一个不等式意味着  $b/M < \frac{2}{3}$ ，而第二个不等式意味着  $b/M > \frac{2}{3}$ ，这是矛盾的。因此，没有能够保证使用小于  $\frac{4}{3}$  的最优装箱数完成装箱的联机算法。 ■

有三种简单算法保证最多使用两倍的最优装箱数，也有相当多更为复杂的算法能够得到更好的结果。

### 2. 下项适配

大概最简单的算法就是下项适配 (next fit) 算法了。当处理任何一项物品时，我们都要检查它是否还能装进刚刚装进物品的那个箱子中去。如果能够装进去，那么就把它放入该箱中；否则，就开辟一个新的箱子。这个算法实现起来非常简单，而且以线性时间运行。图10-21显示了

对于与图10-20相同的输入所得到的装箱过程。

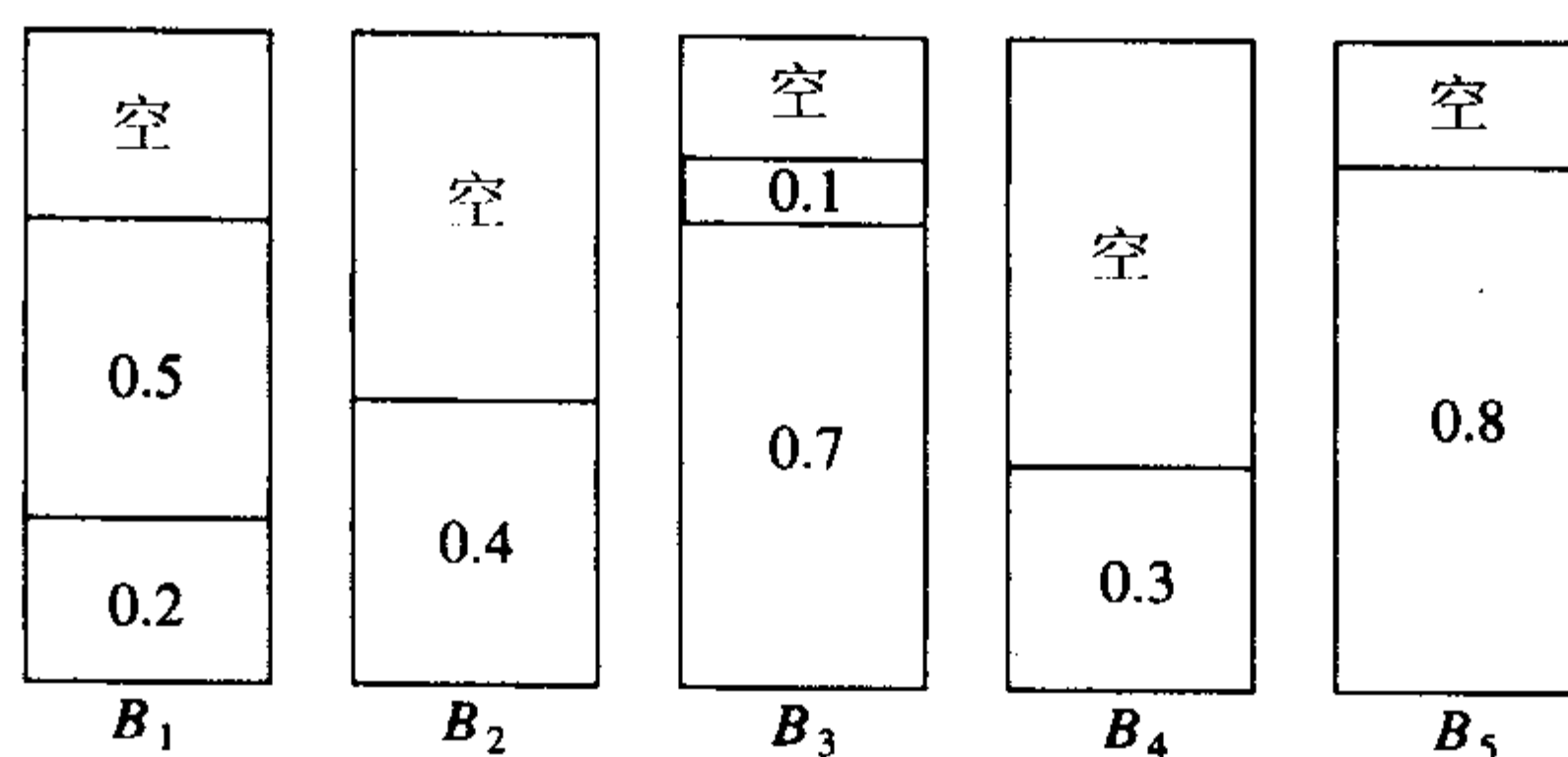


图10-21 对0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8的下项适配算法

下项适配算法不仅编程简单, 而且它的最坏情形的行为也容易分析。

**定理10.2** 令 $M$ 是将一批物品 $I$ 装箱所需的最优装箱数, 则下项适配算法所用的箱子数决不超过 $2M$ 个。存在一些序列使得下项适配算法用箱 $2M-2$ 个。

**证明** 考虑任何相邻的两个箱子 $B_j$ 和 $B_{j+1}$ 。 $B_j$ 和 $B_{j+1}$ 中所有物品的大小之和必然大于1, 否则这些物品就会全部放入 $B_j$ 中。如果将该结果用于所有相邻的两个箱子, 那么可以看到, 最多有一半的空间闲置。因此, 下项适配算法最多使用两倍的最优箱子数。

421

为说明这个界是精确的, 设有 $N$ 项物品, 当 $i$ 是奇数时, 物品的大小 $s_i = 0.5$ , 而当 $i$ 是偶数时,  $s_i = 2/N$ 。设 $N$ 可被4整除。图10-22所示的最优装箱由含有2件大小为0.5的物品的 $N/4$ 个箱子和含有 $N/2$ 件大小为 $2/N$ 的物品的1个箱子组成, 总数为 $(N/4) + 1$ 。图10-23表示下项适配算法使用 $N/2$ 个箱子。因此, 下项适配算法可以用到几乎两倍于最优装箱数的箱子。 ■

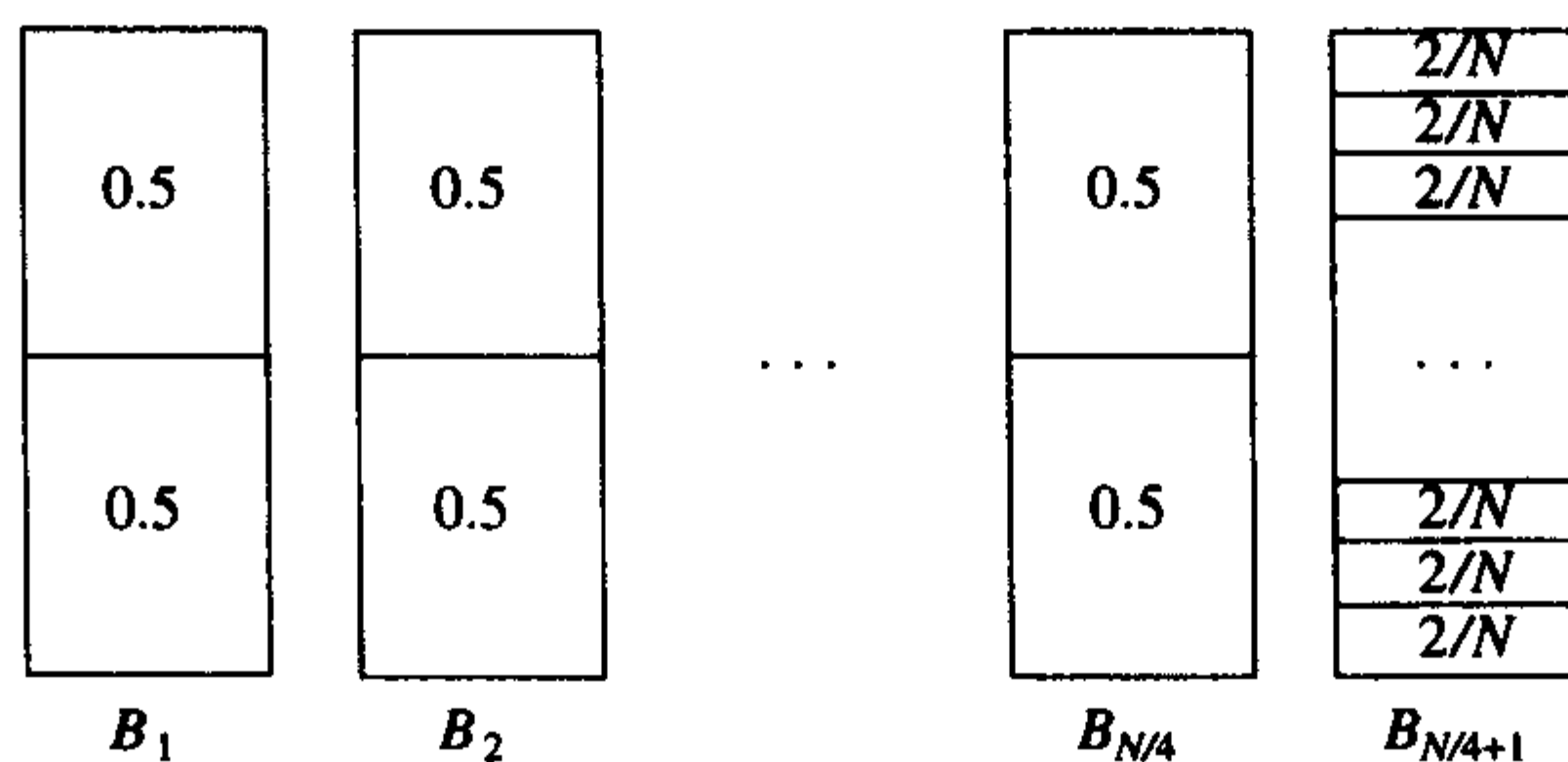


图10-22 对0.5, 2/N, 0.5, 2/N, 0.5, 2/N, ...的最优装箱方法

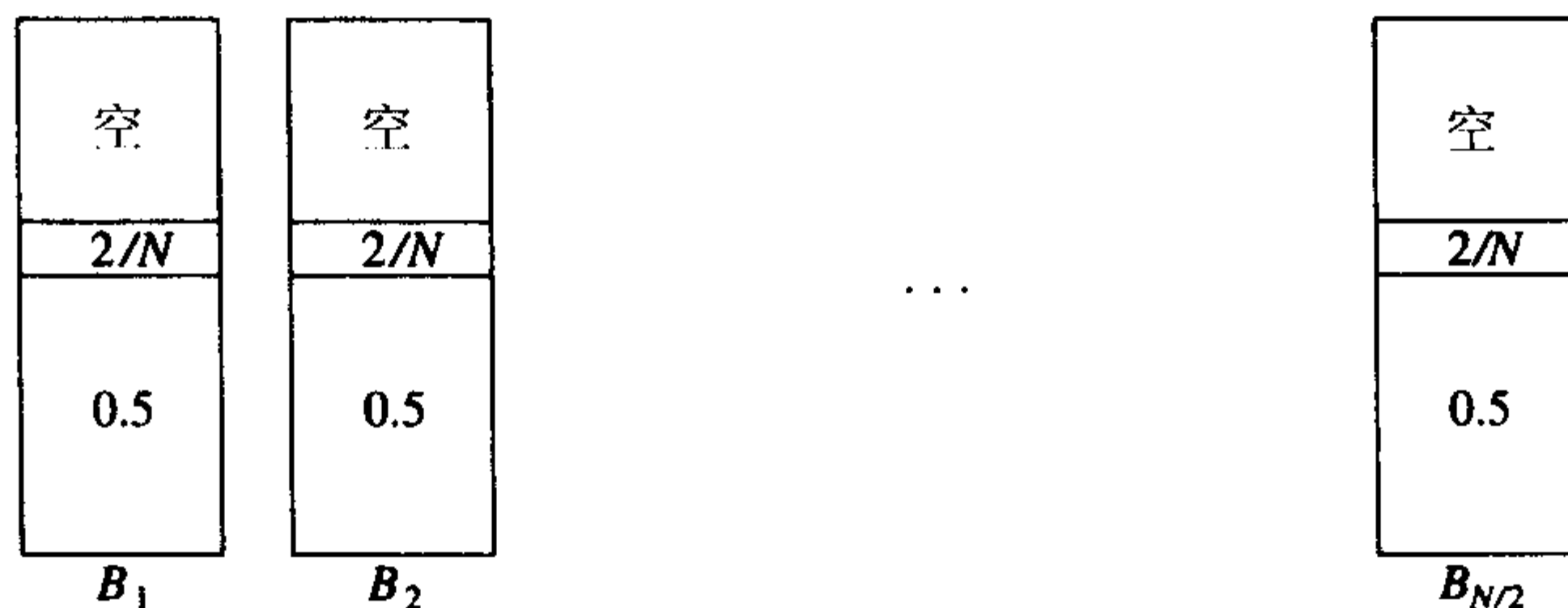


图10-23 对0.5, 2/N, 0.5, 2/N, 0.5, 2/N, ...的下项适配装箱法

### 3. 首次适配

虽然下项适配算法有一个合理的性能保证, 但是, 其效果在实践中却很差, 因为在不需要开辟新箱子的时候它却开辟了新箱子。在前面的样例运行中, 本可以把大小为0.3的物品放入 $B_1$ 或 $B_2$ 而不是开辟一个新箱子。

422 首次适配 (first fit) 算法的策略是依序扫描这些箱子并把一项新的物品放入足能盛下它的第一个箱子中。因此, 只有前面放置物品的箱子已经容不下当前物品的时候, 才开辟一个新箱子。图10-24显示了对标准输入进行首次适配的装箱结果。

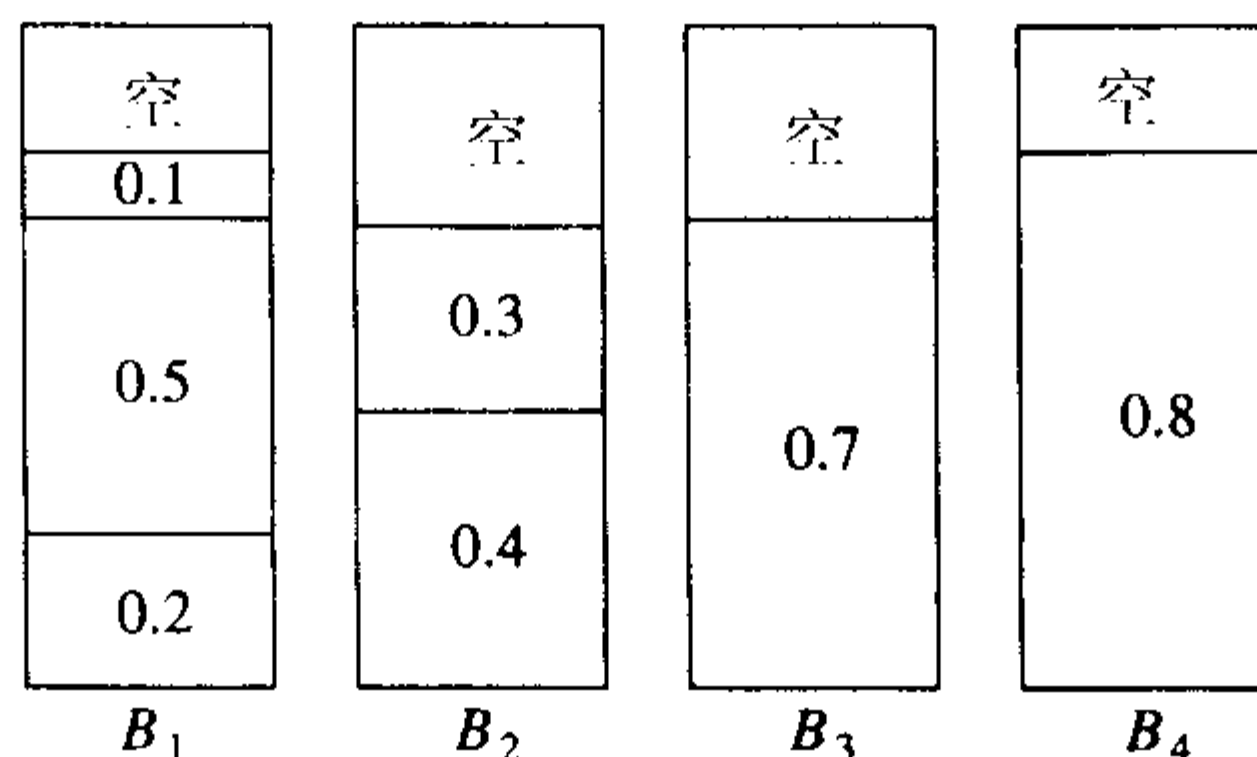


图10-24 对0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8的首次适配装箱

实现首次适配算法的一个简单方法是通过顺序扫描箱子序列处理每一项物品, 这将花费 $O(N^2)$ 时间。有可能以 $O(M \log N)$ 运行来实现首次适配算法; 我们把它留作练习。

略加思索即可明白, 在任一时刻最多有一个箱子其空出的部分大于箱子的一半, 因为若有第二个这样的箱子, 则它装的物品就会装到第一个这样的箱子中了。因此可以断言: 首次适配算法最多使用两倍的最优装箱数。

另一方面, 证明下项适配算法性能的界时所用到的最坏情形对首次适配算法不适用。因此, 人们可能要问: 是否能够证明更好的界呢? 答案是肯定的, 不过证明要复杂一些。

**定理10.3** 令 $M$ 是将一批 $I$ 个物品装箱所需要的最优箱子数, 则首次适配算法使用的箱子数决不多于 $\left\lceil \frac{17}{10}M \right\rceil$ 。存在使得首次适配算法使用 $\left\lceil \frac{17}{10}(M-1) \right\rceil$ 个箱子的序列。

**证明** 参阅本章末尾的参考文献。 ■

使用首次适配算法得出的结果和前面定理指出的结果几乎一样差的例子见图10-25所示。图中的输入由 $6M$ 个大小为 $\frac{1}{7} + \epsilon$ 的项后跟 $6M$ 个大小为 $\frac{1}{3} + \epsilon$ 的项以及接续其后的 $6M$ 个大小为 $\frac{1}{2} + \epsilon$ 的项组成。一种简单的装箱办法是将每种大小的各一项物品装到一个箱子中, 总共需要 $6M$ 个箱子。如果用首次适配算法, 则需要 $10M$ 个箱子。

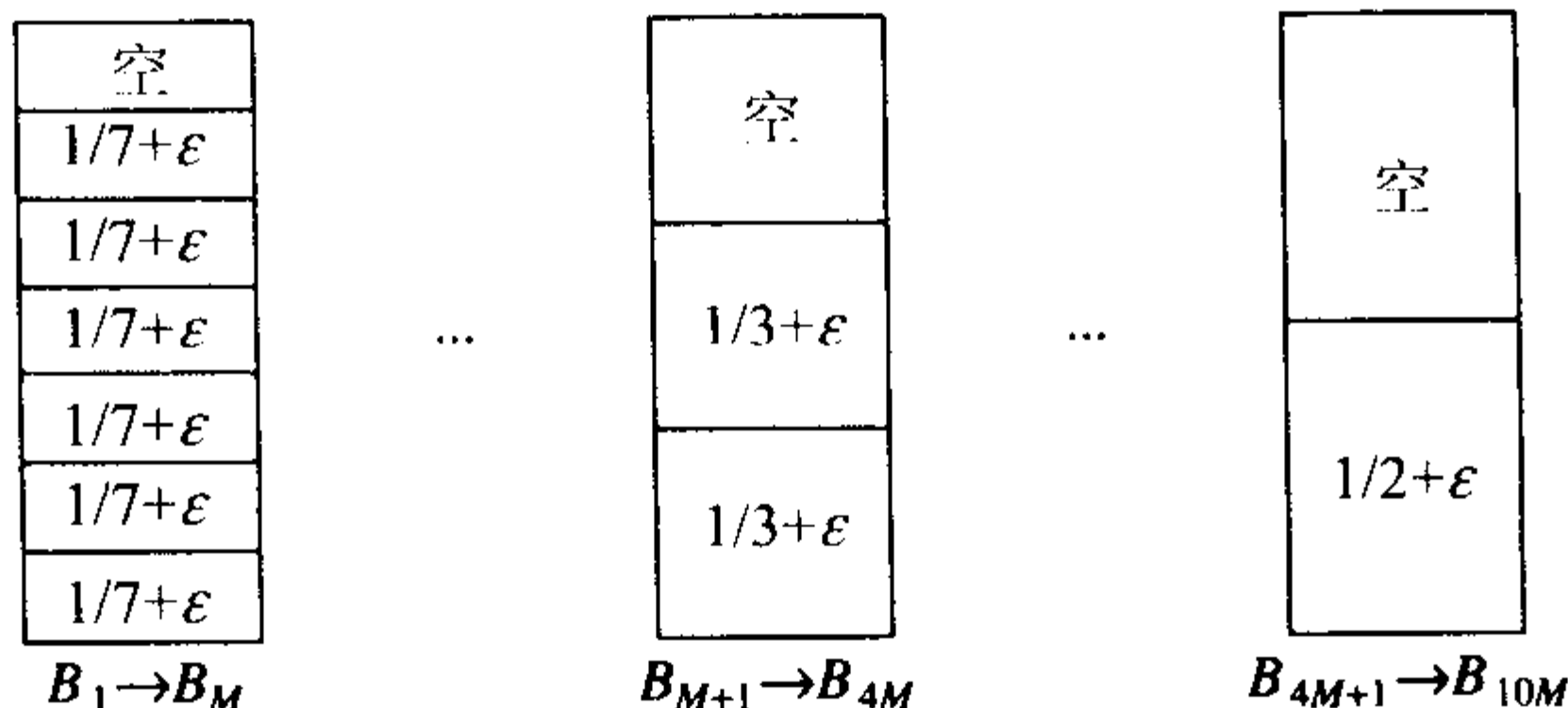


图10-25 首次适配算法使用 $10M$ 个而不是 $6M$ 个箱子的情形

当首次适配算法对大量其大小均匀分布在 $0 \sim 1$ 之间的物品进行运算时, 经验结果指出, 首次适配算法用到大约比最优装箱法多2%的箱子。在许多情况下, 这是完全可以接受的。

#### 4. 最佳适配

我们将要考察的第三种联机策略是**最佳适配** (best fit) 算法。该算法不是把一项新物品放入所发现的第一个能够容纳它的箱子，而是放到所有箱子中能够容纳它的最满的箱子中。典型的装箱方法如图10-26所示。

423

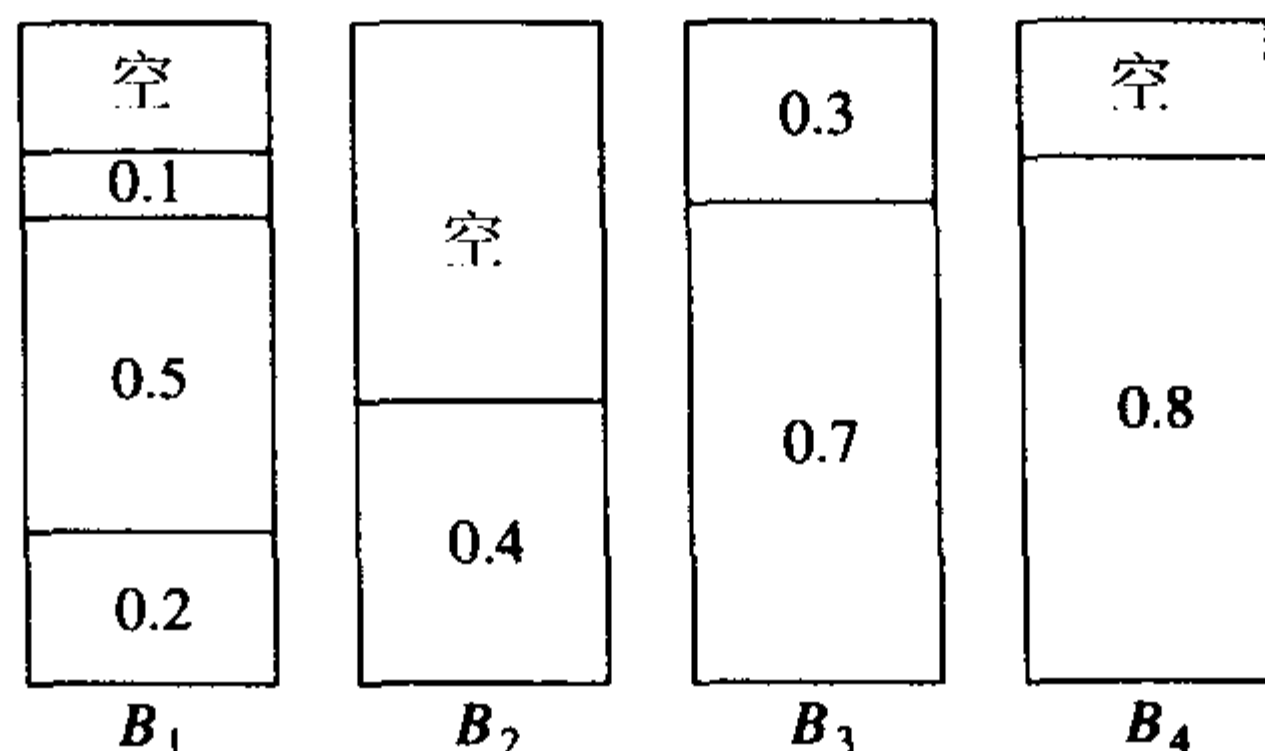


图10-26 对0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8的最佳适配算法

注意，大小为0.3的项不是放在 $B_2$ 中而是放在了 $B_3$ 中，此时它正好把 $B_3$ 填满。由于现在对箱子进行更细致的选择，因此人们可能认为算法性能保证会有所改善。但是情况并非如此，因为一般的不好情形都是相同的。最佳适配算法决不会超过最优算法的大约1.7倍，而且存在一些输入，对于这些输入该算法（几乎）达到这个界。不过，最佳适配算法编程还是简单的，特别是当需要 $O(\text{Mlog}N)$ 算法的时候，而且该算法对随机的输入确实表现得更好。

#### 5. 脱机算法

如果能够观察全部物品以后再算出答案，那么应该会做得更好。事实确实如此，由于通过彻底的搜索最终能够找到最优装箱方法，因此对联机情形就已经有了一个理论上的改进。

所有联机算法的主要问题在于，将大项物品装箱困难，特别是当它们在输入的后期出现的时候。围绕这个问题的自然方法是将各项物品排序，把最大的物品放在最先。此时可以应用首次适配算法或最佳适配算法，分别得到**首次适配递减** (first fit decreasing) 算法和**最佳适配递减** (best fit decreasing) 算法。图10-27指出在我们的例子中这会产生最优解（尽管在一般情形下显然未必会如此）。

424

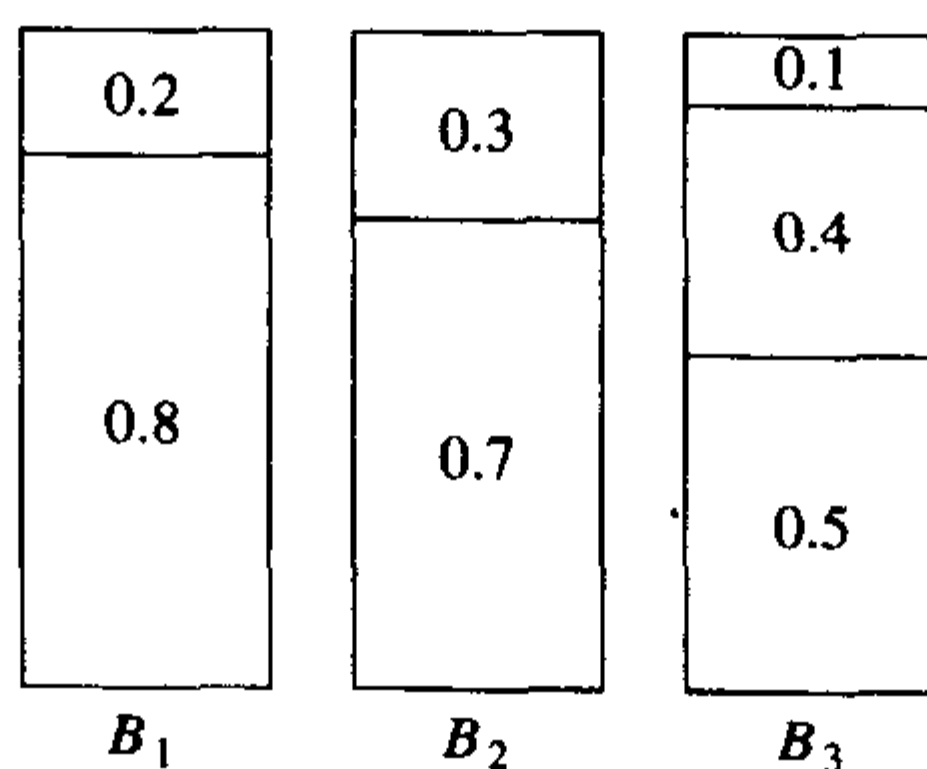


图10-27 对0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1的首次适配算法

本节我们将讨论首次适配递减算法。对于最佳适配递减算法，结果几乎是一样的。由于存在物品大小不互异的可能，因此有些作者更愿意把首次适配递减算法叫作**首次适配非增** (first fit nonincreasing) 算法。我们将沿用原始的名称。不失一般性，假设输入数据已经根据大小排序。

我们能够做的第一个评注是，首次适配算法使用 $10M$ 个而不是 $6M$ 个箱子的不好情形在物品项被排序的情况下不会再发生。我们将证明，如果一种最优装箱法使用 $M$ 个箱子，那么首次适配递减算法使用的箱子数决不超过 $(4M + 1)/3$ 。



这个结果依赖于两项观察。首先，所有重量大于 $\frac{1}{3}$ 的项将被放入前 $M$ 个箱子内。这意味着，在这 $M$ 个箱子之外的其余箱子中所有各项的重量至多是 $\frac{1}{3}$ 。其次，在其余箱子中最多有 $M-1$ 项物品。把这两个结果结合起来可以发现，其余的箱子最多需要 $\lceil (M-1)/3 \rceil$ 个。下面证明这两个观察结果。

**引理10.1** 令 $N$ 项物品的输入大小（以递减顺序排序）分别为 $s_1, s_2, \dots, s_N$ ，并设最优装箱方法使用 $M$ 个箱子。那么，首次适配递减算法放到 $M$ 个箱子之外的其余箱子中的所有物品的大小最多为 $\frac{1}{3}$ 。

**证明** 设第 $i$ 项物品是放入第 $M+1$ 个箱子中的第一项，需要证明 $s_i \leq \frac{1}{3}$ 。我们将使用反证法证明这个结论。设 $s_i > \frac{1}{3}$ 。 ■

由于这些物品的大小是以递减顺序排列的，因此， $s_1, s_2, \dots, s_{i-1} > \frac{1}{3}$ 。由此得知，所有的箱子 $B_1, B_2, \dots, B_M$ 每个最多有两项物品。

考虑在第 $i-1$ 项物品被放入一个箱子后但第 $i$ 项物品尚未放入时系统的状态。现在我们要证明（在 $s_i > \frac{1}{3}$ 的假设下）前 $M$ 个箱子的排列如下：首先是有些箱子内恰好有一项物品，然后是其余的箱子内有两项物品。

425 设有两个箱子 $B_x$ 和 $B_y$ ， $1 \leq x < y \leq M$ ， $B_x$ 有两项而 $B_y$ 有一项。令 $x_1$ 和 $x_2$ 是 $B_x$ 中的两项物品，并令 $y_1$ 是 $B_y$ 中的一项物品。 $x_1 \geq y_1$ ，因为 $x_1$ 被放在较前的箱子中。根据类似的推理， $x_2 \geq s_i$ 。因此， $x_1 + x_2 \geq y_1 + s_i$ 。这意味着， $s_i$ 应该可以放在 $B_y$ 中。根据假设，这是不可能的。因此，如果 $s_i > \frac{1}{3}$ ，那么在我们试图处理 $s_i$ 时，这样安排前 $M$ 个箱子，使得前 $j$ 个箱子各装一项物品，而后 $M-j$ 个箱子各放两项物品。

为了证明该引理，我们将证明不存在将所有物品装入 $M$ 个箱子的方法，这和引理的假设矛盾。

显然，在 $s_1, s_2, \dots, s_j$ 中使用任何算法都没有两项可以放入一个箱子中，因为如果能放，那么首次适配算法也能放。我们还知道，首次适配算法尚未把大小为 $s_{j+1}, s_{j+2}, \dots, s_i$ 的任一项放入前 $j$ 个箱子中，因此它们都不能再往前 $j$ 个箱子中放。这样，在任何装箱方法中，特别是最优装箱方法中，必然存在 $j$ 个箱子不包含这些项。由此可知，大小为 $s_{j+1}, s_{j+2}, \dots, s_{i-1}$ 的项必然包含在 $M-j$ 个箱子的集合中，考虑到前面的讨论，于是这些项的总数为 $2(M-j)^1$ 。

注意，如果 $s_i > \frac{1}{3}$ ，那么只要证明 $s_i$ 没有办法放入这 $M$ 个箱子之一中去，该引理的证明也就完成了。事实上，显然它不能放入这 $j$ 个箱子中，因为假如能放入，那么首次适配算法也能够这么做。把它放入剩下的 $M-j$ 个箱子之一中需要把 $2(M-j)+1$ 项物品分发到这 $M-j$ 个箱子中。因此，某个箱子就不得不装入三项物品，而其中的每一项都大于 $\frac{1}{3}$ ，很明显，这是不可能的。

1. 回忆首次适配算法把这些元素装入 $M-j$ 个箱子并在每个箱子中放入两项物品。因此有 $2(M-j)$ 项。

这与所有大小的物品都能够装入 $M$ 个箱子的事实矛盾, 因此开始的假设肯定是不正确的, 从而 $s_i \leq \frac{1}{3}$ 。

**引理10.2** 放入其余箱子中的物品的个数最多是 $M-1$ 。

**证明** 假设放入其余(即附加的)箱子中的物品至少有 $M$ 个。我们知道 $\sum_{i=1}^N s_i \leq M$ , 因为所有的物品都可装入 $M$ 个箱子。设对于 $1 \leq j \leq M$ , 箱子 $B_j$ 装入后总权为 $W_j$ 。设前 $M$ 个其余箱子中的物品大小为 $x_1, x_2, \dots, x_M$ 。此时, 由于前 $M$ 个箱子中的项加上前 $M$ 个其余箱子中的项是所有物品的一个子集, 于是:

$$\sum_{i=1}^N s_i \geq \sum_{j=1}^M W_j + \sum_{j=1}^M x_j \geq \sum_{j=1}^M (W_j + x_j)$$

现在 $W_j + x_j > 1$ , 否则对应于 $x_j$ 的项就已经放入 $B_j$ 中。因此:

$$\sum_{i=1}^N s_i > \sum_{j=1}^M 1 > M$$

但如果 $N$ 项物品可以被装入 $M$ 个箱子中, 这是不可能的。因此, 最多只能有 $M-1$ 项其余的物品。 ■

**定理10.4** 令 $M$ 是将物品集 $I$ 装箱所需的最优箱子数, 则首次适配递减算法所用的箱子数绝不超过 $(4M+1)/3$ 。 426

**证明** 存在 $M-1$ 项其余箱子中的物品, 其大小至多为 $\frac{1}{3}$ 。因此, 最多可能存在 $\lceil (M-1)/3 \rceil$ 个其余的箱子。从而, 首次适配递减算法使用的箱子总数最多为 $\lceil (4M-1)/3 \rceil \leq (4M+1)/3$ 。 ■

可以证明, 首次适配递减算法和下项适配递减算法都有一个紧得多的界。

**定理10.5** 令 $M$ 是将物品集 $I$ 装箱所需的最优箱子数, 则首次适配递减算法所用的箱子数绝不超过 $\frac{11}{9}M + 4$ 。此外, 存在使得首次适配递减算法用到 $\frac{11}{9}$ 个箱子的序列。

**证明** 上界需要非常复杂的分析。下界可以通过下述序列展示: 先是大小为 $\frac{1}{2} + \varepsilon$ 的 $6M$ 项物品, 其后是大小为 $\frac{1}{4} + 2\varepsilon$ 的 $6M$ 项物品, 接着是大小为 $\frac{1}{4} + \varepsilon$ 的 $6M$ 项物品, 最后是大小为 $\frac{1}{4} - 2\varepsilon$ 的 $12M$ 项物品。图10-28指出最优装箱需要 $9M$ 个箱子, 而首次适配递减算法需要 $11M$ 个箱子。 ■

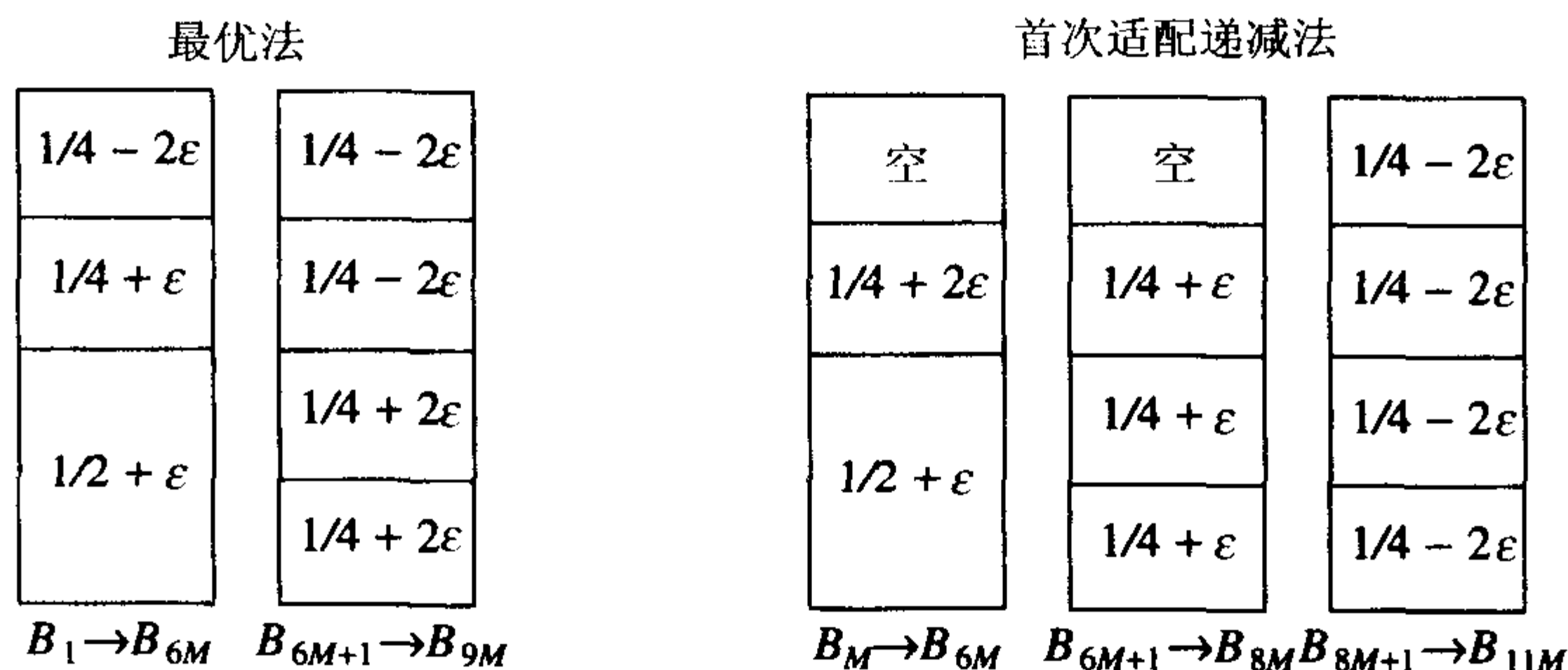


图10-28 首次适配递减算法使用 $11M$ 个箱子但只有 $9M$ 个箱子就足够完成装箱的例子

在实践中，首次适配递减算法的效果非常好。如果大小在单位区间内均匀分布，那么多用的箱子的期望个数为  $\Theta(\sqrt{M})$ 。装箱算法是简单贪心试探算法能够给出好结果的一个很好的例子。

## 10.2 分治算法

427 用于设计算法的另一种常用技巧为分治 (divide and conquer)。分治算法由两部分组成：

- 分 (divide)：递归解决较小的问题（当然，除基本情形外）。
- 治 (conquer)：然后，从子问题的解构建原问题的解。

习惯上，在正文中至少含有两个递归调用的例程叫作分治算法，而正文中只含一个递归调用的例程不是分治算法。我们一般认为子问题是不相交的（即基本上不重叠）。下面回顾一下本书已涉及的某些递归算法。

我们已经看到几个分治算法。在2.4.3节见过最大子序列和问题的一个  $O(M \log N)$  解。在第4章，看到一些线性时间的树遍历方法。在第7章，见过分治算法的经典例子，即归并排序和快速排序，它们分别有  $O(M \log N)$  的最坏情形以及平均情形的时间界。

我们还看到过递归算法的若干例子，在分类上它们很可能不算作分治算法，而只是化简到一个更简单的情况。在1.3节，看到一个简单的显示数的例程。在第2章，使用递归执行有效的取幂运算。在第4章，考察了二叉查找树的一些简单搜索例程。在6.6节，见过用于合并左式堆的简单的递归。7.7节给出了一个花费线性平均时间解决选择问题的算法。第8章递归地写出了不相交集的 find 操作。第9章给出以 Dijkstra 算法重新找出最短路径的一些例程以及对图进行深度优先搜索的其他过程。这些算法实际上都不是分治算法，因为只进行了一次递归调用。

我们在2.4节还看到计算斐波那契数的非常差的递归例程。可以称其为分治算法，但它的效率太低了，因为实际上问题根本没有被分割。

在这一节，我们将看到分治算法更多的范例。第一个应用是计算几何中的问题。给定平面上的  $N$  个点，我们将证明最近的一对点可以在  $O(M \log N)$  时间找到。本章后面的一些练习描述了计算几何中的另外一些问题，它们可以由分治算法求解。本节其余部分介绍理论上极其有趣的一些结果，提供一个算法以  $O(N)$  最坏情形时间解决选择问题，还要证明可以用  $o(N^2)$  操作将2个  $N$  位的数相乘并以  $o(N^3)$  操作将两个矩阵相乘。可是，虽然这些算法最坏情形时间界比传统算法要好，但是在输入量不大的情况下，它们并不实用。

### 10.2.1 分治算法的运行时间

我们将要看到的所有有效的分治算法都是把问题分成一些子问题，每个子问题都是原问题的一部分，然后进行某些附加的工作以算出最后的答案。作为一个例子，我们已经看到归并排序对两个问题进行运算，每个问题均为原问题大小的一半，然后用到  $O(N)$  附加工作。由此得到运行时间方程（带有适当的初始条件）：

428 
$$T(N) = 2T(N/2) + O(N)$$

由第7章可知，该方程的解为  $O(M \log N)$ 。下面的定理可以用来确定大部分分治算法的运行时间。

**定理10.6** 方程  $T(N) = aT(N/b) + \Theta(N^k)$  的解为

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{若 } a > b^k \\ O(N^k \log N) & \text{若 } a = b^k \\ O(N^k) & \text{若 } a < b^k \end{cases}$$

其中  $a \geq 1$ ,  $b > 1$ 。

**证明** 根据第7章归并排序的分析, 假设  $N$  是  $b$  的幂; 于是, 可令  $N = b^m$ 。此时  $N/b = b^{m-1}$  及  $N^k = (b^m)^k = b^{mk} = b^{km} = (b^k)^m$ 。假设  $T(1) = 1$ , 并忽略  $\Theta(N^k)$  中的常数因子, 则有

$$T(b^m) = aT(b^{m-1}) + (b^k)^m$$

如果用  $a^m$  除两边, 则得到方程

$$\frac{T(b^m)}{a^m} = \frac{T(b^{m-1})}{a^{m-1}} + \left\{ \frac{b^k}{a} \right\}^m \quad (10-3)$$

我们可以该方程用于  $m$  的其他值, 得到

$$\frac{T(b^{m-1})}{a^{m-1}} = \frac{T(b^{m-2})}{a^{m-2}} + \left\{ \frac{b^k}{a} \right\}^{m-1} \quad (10-4)$$

$$\frac{T(b^{m-2})}{a^{m-2}} = \frac{T(b^{m-3})}{a^{m-3}} + \left\{ \frac{b^k}{a} \right\}^{m-2} \quad (10-5)$$

$\vdots$

$$\frac{T(b^1)}{a^1} = \frac{T(b^0)}{a^0} + \left\{ \frac{b^k}{a} \right\}^1 \quad (10-6)$$

使用将式 (10-3) 到 (10-6) 叠缩方程两边分别加起来的标准技巧, 等号左边的所有项实际上与等号右边的前一项相消, 由此得到

$$\frac{T(b^m)}{a^m} = 1 + \sum_{i=1}^m \left\{ \frac{b^k}{a} \right\}^i \quad (10-7)$$

$$= \sum_{i=0}^m \left\{ \frac{b^k}{a} \right\}^i \quad (10-8)$$

因此

$$T(N) = T(b^m) = a^m \sum_{i=0}^m \left\{ \frac{b^k}{a} \right\}^i \quad (10-9) \quad \boxed{429}$$

如果  $a > b^k$ , 那么和就是一个公比小于1的几何级数。由于无穷级数的和收敛于一个常数, 因此该有限的和也以一个常数为界, 从而式 (10-10) 成立:

$$T(N) = O(a^m) = O(a^{\log_b N}) = O(N^{\log_b a}) \quad (10-10)$$

如果  $a = b^k$ , 那么和中的每一项均为1。由于和含有  $1 + \log_b N$  项而  $a = b^k$  意味着  $\log_b a = k$ , 于是

$$\begin{aligned} T(N) &= O(a^m \log_b N) = O(N^{\log_b a} \log_b N) = O(N^k \log_b N) \\ &= O(N^k \log N) \end{aligned} \quad (10-11)$$

最后, 如果  $a < b^k$ , 那么该几何级数中的项都大于1, 且1.2.3节中的第二个公式成立。有

$$T(N) = a^m \frac{(b^k/a)^{m+1} - 1}{(b^k/a) - 1} = O(a^m (b^k/a)^m) = O((b^k)^m) = O(N^k) \quad (10-12)$$

定理的最后一种情形得证。 ■

作为一个例子, 归并排序有  $a = b = 2$  且  $k = 1$ 。第二种情形成立, 因此答案为  $O(M \log N)$ 。如果求解三个问题, 每个问题都是原始大小的一半, 使用  $O(N)$  的附加工作将解合并起来, 则  $a = 3$ ,  $b = 2$  而  $k = 1$ 。此处第一种情形成立, 于是得到界  $O(N^{\log_2 3}) = O(N^{1.59})$ 。一个求解三个一半大小的问题



但需要花费 $O(N^2)$ 时间以合并解的算法，其运行时间将是 $O(N^2)$ ，因为此时第三种情形成立。

有两个重要的情形定理10.6没有包括。下面再叙述两个定理，但把证明留作练习。定理10.7推广了前面的定理。

**定理10.7** 方程 $T(N) = aT(N/b) + \Theta(N^k \log^p N)$ 的解为

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{若 } a > b^k \\ O(N^k \log^{p+1} N) & \text{若 } a = b^k \\ O(N^k \log^p N) & \text{若 } a < b^k \end{cases}$$

其中 $a \geq 1$ ,  $b > 1$ 且 $p \geq 0$ 。

**定理10.8** 如果 $\sum_{i=1}^k \alpha_i < 1$ ，则方程 $T(N) = \sum_{i=1}^k T(\alpha_i N) + O(N)$ 的解为 $T(N) = O(N)$ 。

## 10.2.2 最近点问题

我们的第一个问题的输入是平面上的点集 $P$ 。如果 $p_1 = (x_1, y_1)$ 和 $p_2 = (x_2, y_2)$ ，那么 $p_1$ 和 $p_2$ 间的欧几里得距离为 $[(x_1 - x_2)^2 + (y_1 - y_2)^2]^{1/2}$ 。我们要找出一对最近的点。有可能两个点位于同一个位置；在这种情形下这两个点就是最近的，它们的距离为零。

**430** 如果存在 $N$ 个点，那么就存在 $N(N-1)/2$ 对点间的距离。可以检查所有这些距离，得到一个很短的程序，不过这是一个花费 $O(N^2)$ 的算法。由于这种方法是一种穷尽的搜索，因此应该期望做得更好一些。

假设平面上这些点已经按照 $x$ 的坐标排过序，这最多只是在最终的时间界上多加了 $O(M \log N)$ 。由于将证明整个算法的 $O(M \log N)$ 界，因此从复杂度的观点来看，该排序基本上没增加时间消耗的级别。

图10-29画出一个小的样本点集 $P$ 。既然这些点已按 $x$ 坐标排序，那么就可以划一条想像的垂线，把点集分成两半： $P_L$ 和 $P_R$ 。这做起来相当简单。现在得到的情形几乎和我们在2.4.3节的最大子序列和问题中见过的情形完全相同。最近的一对点或者都在 $P_L$ 中，或者都在 $P_R$ 中，或者一个在 $P_L$ 中而另一个在 $P_R$ 中。把这三个距离分别叫作 $d_L$ 、 $d_R$ 和 $d_C$ 。图10-30给出点集的划分和这三个距离。

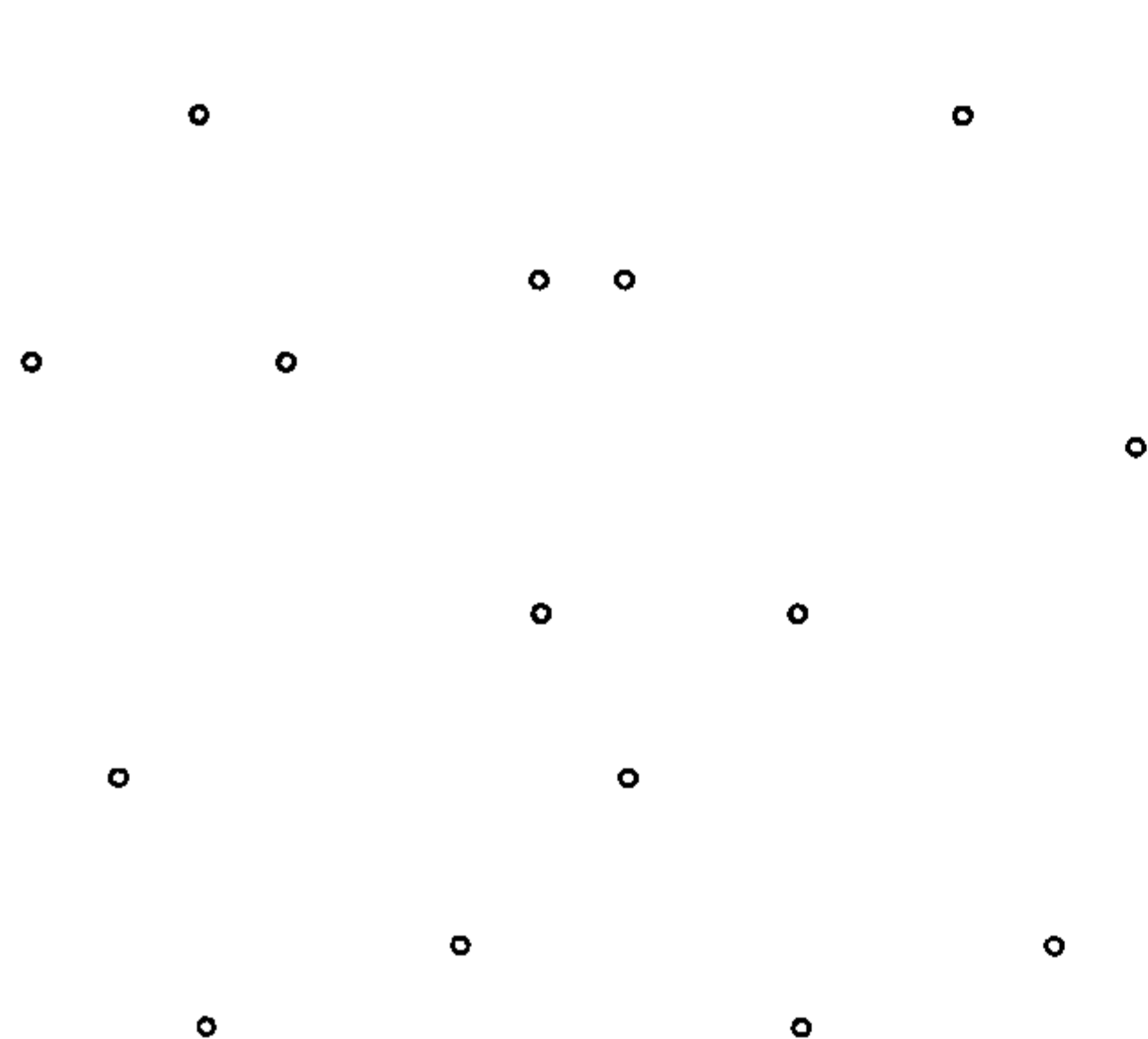


图10-29 一个小规模的点集

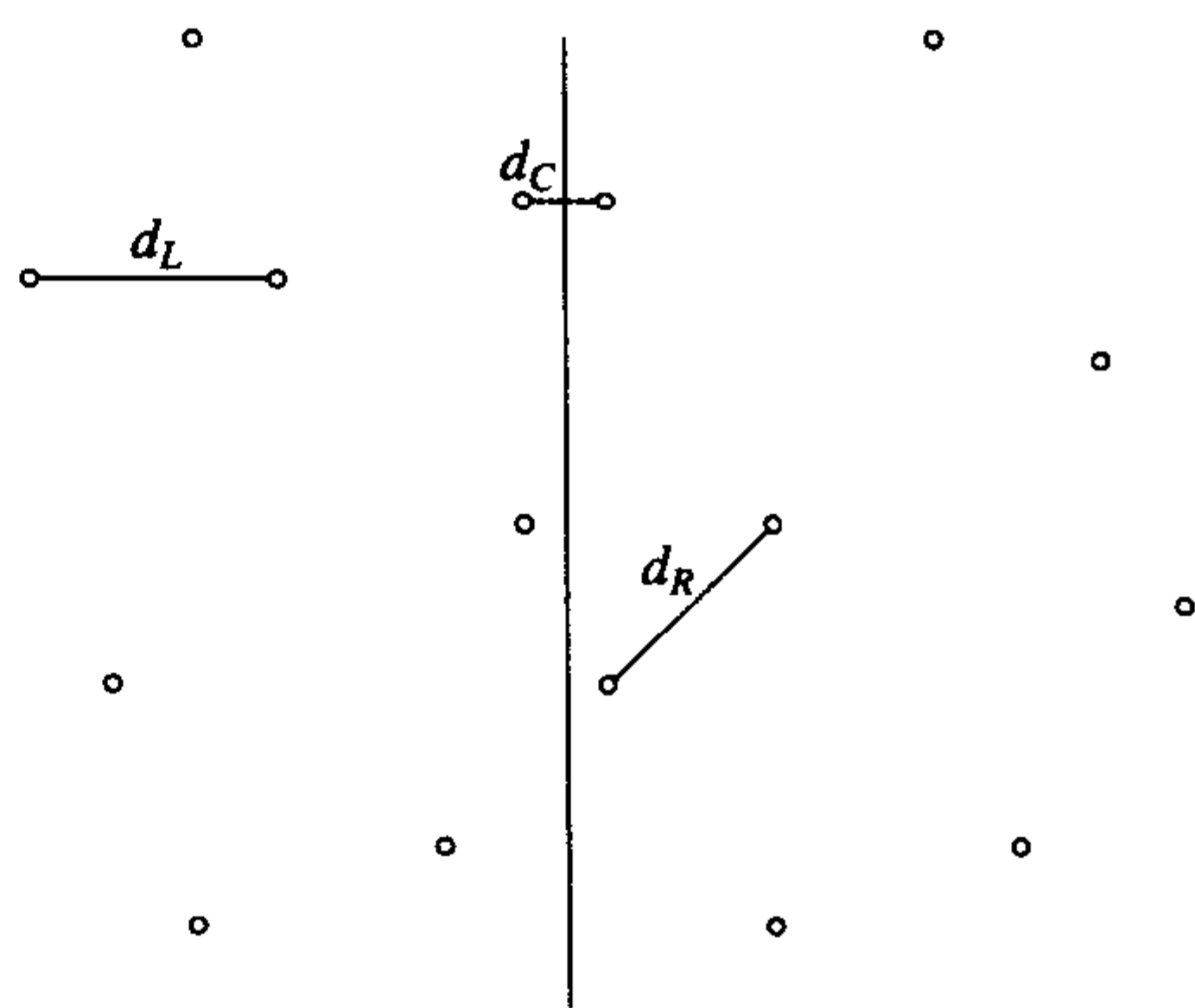


图10-30 被分成 $P_L$ 和 $P_R$ 的点集 $P$ ；显示了最短的距离

我们可以递归地计算 $d_L$ 和 $d_R$ 。接下来的问题就是计算 $d_C$ 。由于我们期望得到一个 $O(M \log N)$ 的解，因此必须能够仅仅多花 $O(N)$ 的附加工作就计算出 $d_C$ 。我们已经知道，如果一个过程由两个一半大小的递归调用和附加的 $O(N)$ 工作组成，那么总的时间将是 $O(M \log N)$ 。

令  $\delta = \min(d_L, d_R)$ 。第一个观察结论是，如果  $d_C$  对  $\delta$  有所改进，那么只需计算  $d_C$ 。如果  $d_C$  是这样的距离，则决定  $d_C$  的两个点必然在分割线的  $\delta$  距离之内；把这个区域叫作带 (strip)。如图10-31所示，这个观察结论限定了需要考虑的点的个数（此例中的  $\delta = d_R$ ）。

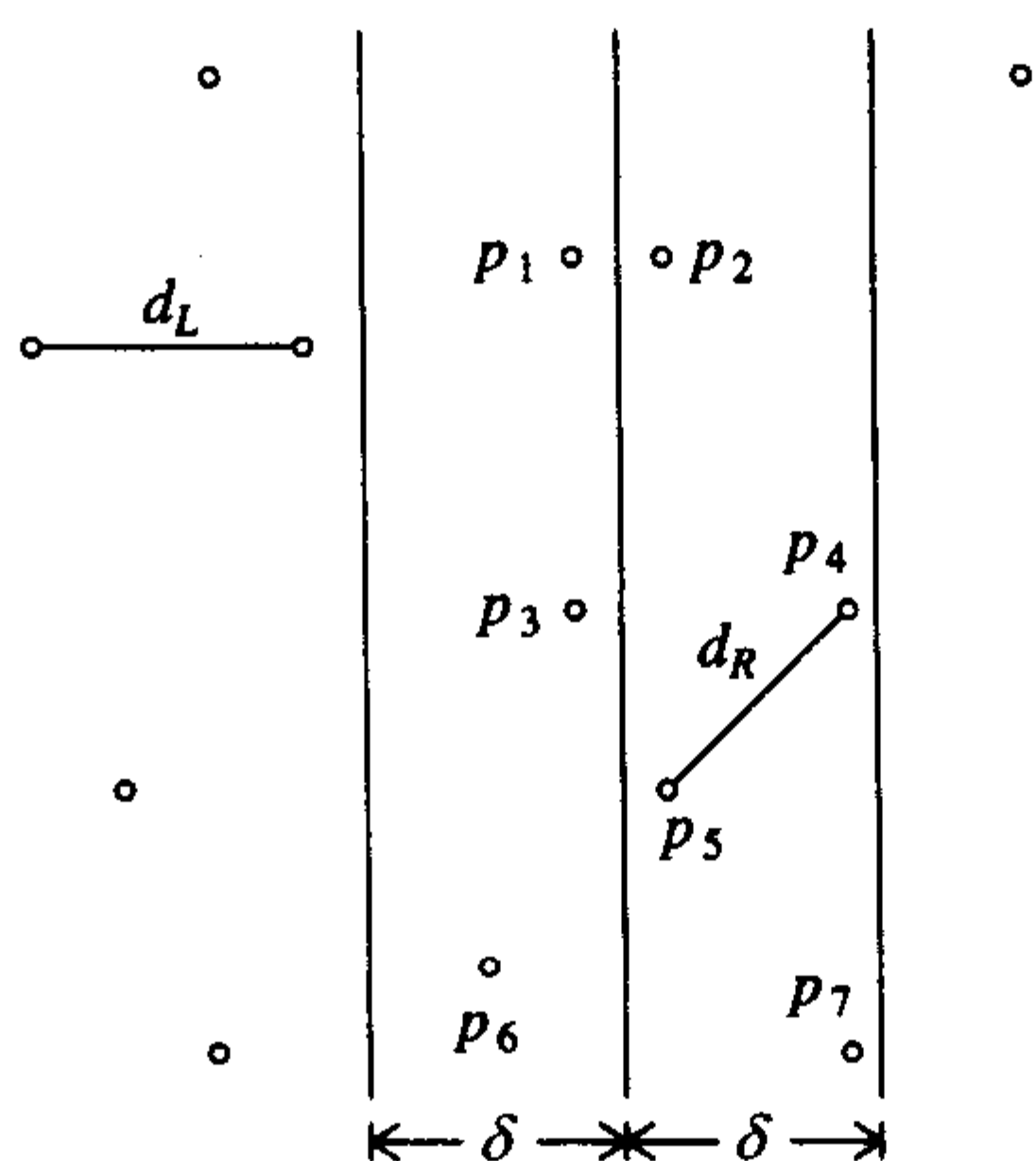


图10-31 双道带区域，包含对于  $d_C$  带所考虑的全部点

有两种策略可以用来计算  $d_C$ 。对于均匀分布的大型点集，预计位于该带中的点的个数是非常少的。事实上，容易论证平均只有  $O(\sqrt{N})$  个点在这个带中。因此，可以以  $O(N)$  时间对这些点进行蛮力计算。图10-32中的伪代码实现了该策略，其中按照C++语言的约定，点的下标从0开始。

431

```
// Points are all in the strip
for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( dist( p_i, p_j ) < delta )
            delta = dist( p_i, p_j );
```

图10-32  $\min(\delta, d_C)$  的蛮力计算

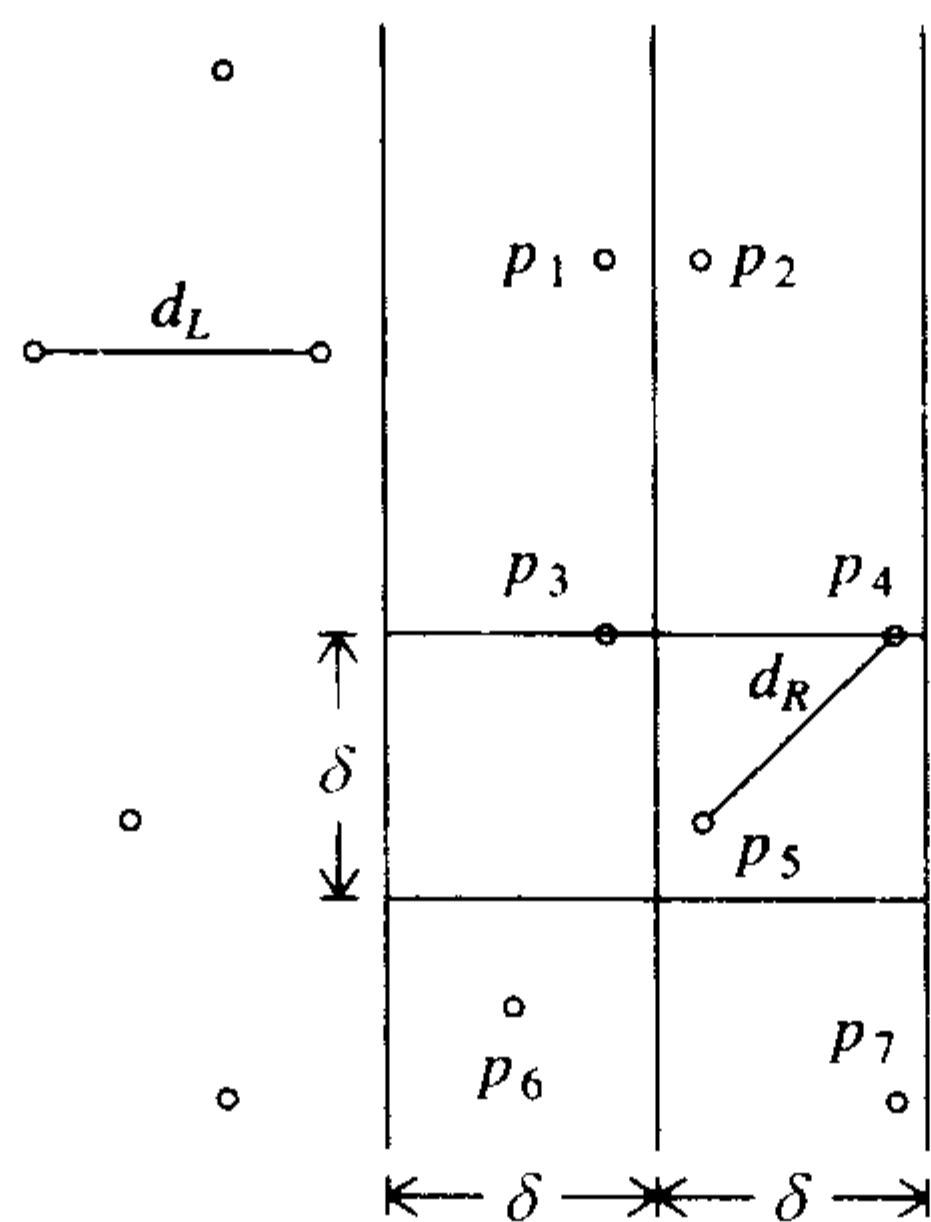
在最坏情形下，所有的点可能都在这条带状区域内，因此这种方法不总能以线性时间运行。可以用下列的观察结果改进这个算法：确定  $d_C$  的两个点的  $y$  坐标相差最多是  $\delta$ 。否则， $d_C > \delta$ 。设带中的点按照它们的  $y$  坐标排序。因此，如果  $p_i$  和  $p_j$  的  $y$  坐标相差大于  $\delta$ ，那么可以继续处理  $p_{i+1}$ 。这个简单的修改在图10-33中实现。

432

```
// Points are all in the strip and sorted by y-coordinate
for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( p_i and p_j's y-coordinates differ by more than delta )
            break; // Go to next p_i.
        else
            if( dist( p_i, p_j ) < delta )
                delta = dist( p_i, p_j );
```

图10-33  $\min(\delta, d_C)$  的优化计算

这个附加的测试对运行时间有着显著的影响，因为对于每一个  $p_i$ ，在  $p_i$  和  $p_j$  的  $y$  坐标相差大于  $\delta$  并被迫退出内层 `for` 循环以前，只有少数的点  $p_j$  被考察。例如，图10-34显示了对于点  $p_3$  只有两个点  $p_4$  和  $p_5$  落在垂直距离  $\delta$  之内的带状区域中。

图10-34 在第2个for循环中只考虑 $p_4$ 和 $p_5$ 

对于任意的点 $p_i$ ，在最坏情形下最多有7个点 $p_j$ 要考虑。这是因为这些点必定落在该带状区域左半部分的 $\delta \times \delta$ 方块内，或者落在该带状区域右半部分的 $\delta \times \delta$ 方块内。另一方面，在每个 $\delta \times \delta$ 方块内的所有的点至少分离 $\delta$ 。在最坏情形下，每个方块包含4个点，每个角上一个点。这些点中有一个是 $p_i$ ，最多还剩下7个点要考虑。最坏情形的状况见图10-35所示。注意，虽然 $p_{L2}$ 和 $p_{R1}$ 有相同的坐标，但它们可以是不同的点。对于实际的分析来说，唯一重要的是 $\lambda \times 2\lambda$ 的矩形区域中的点的个数为 $O(1)$ ，这当然很清楚。

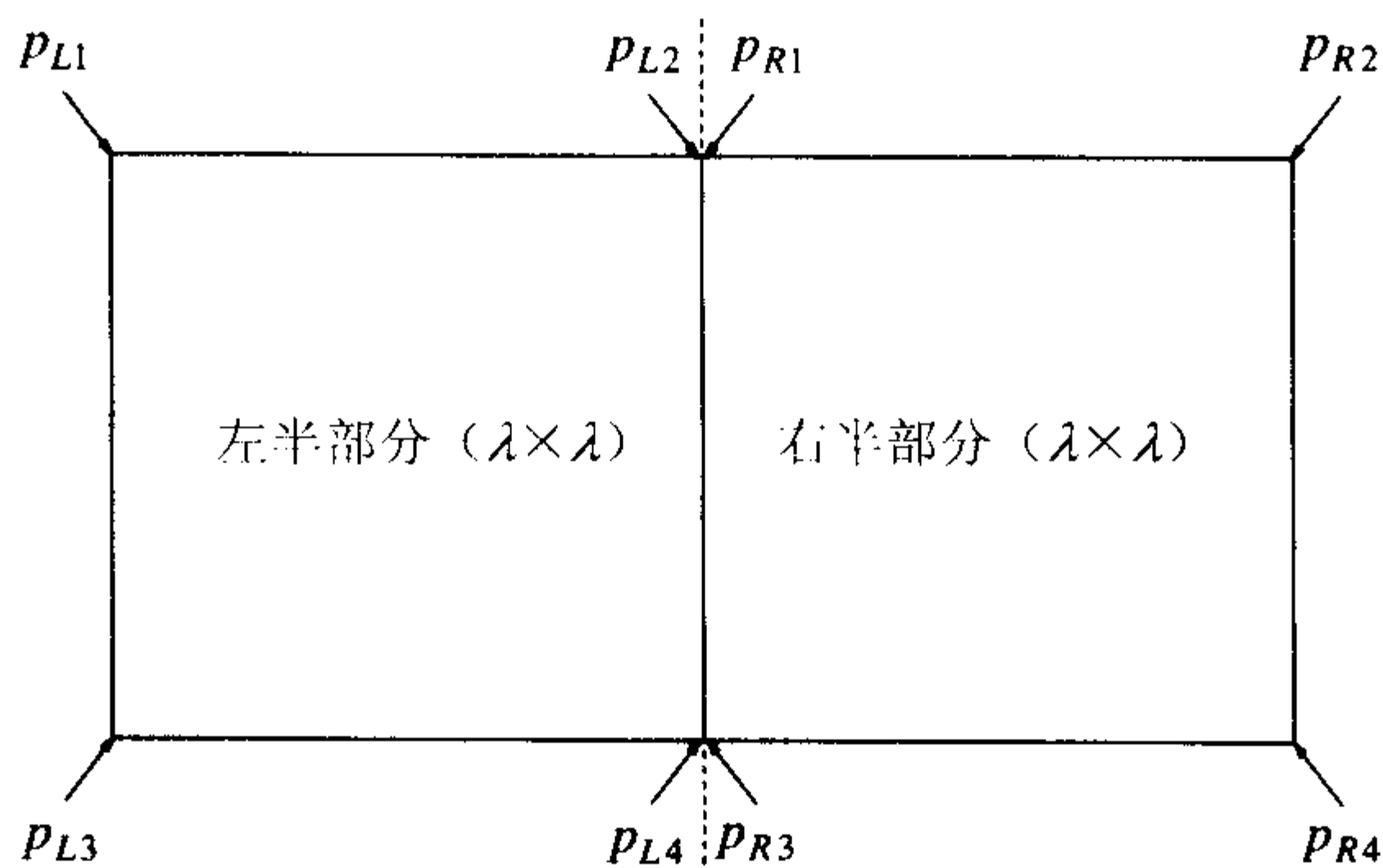


图10-35 最多有8个点在该矩形中，其中有两个坐标由两个点共享

因为对于每个 $p_i$ 最多有7个点要考虑，所以计算比 $\delta$ 好的 $d_c$ 的时间是 $O(N)$ 。因此，基于两个一半大小的递归调用加上合并两个结果的线性附加工作，看来最近点问题似乎有一个 $O(M \log N)$ 解。但是，我们还没有真正得到 $O(M \log N)$ 的解。

问题在于，我们已经假设这些点按照 $y$ 坐标排序是现成的。如果对于每个递归调用都执行这种排序，那么又有 $O(M \log N)$ 的附加工作：这就得到一个 $O(M \log^2 N)$ 算法。不过问题不全这么糟，尤其在和蛮力 $O(N^2)$ 算法比较的时候。然而，不难把对于每个递归调用的工作简化到 $O(N)$ ，从而保证 $O(M \log N)$ 算法。

433

我们将保留两个表。一个是按照 $x$ 坐标排序的点的表，而另一个是按照 $y$ 坐标排序的点的表。分别称这两个表为 $P$ 和 $Q$ 。这两个表可以通过一个预处理排序步骤花费 $O(M \log N)$ 得到，因此并不影响时间界。 $P_L$ 和 $Q_L$ 是传递给左半部分递归调用的参数表， $P_R$ 和 $Q_R$ 是传递给右半部分递归调用的参数表。我们已经看到， $P$ 很容易在中间分开。一旦分割线已知，我们依序转到 $Q$ ，把每一个

元素放入相应的 $Q_L$ 或 $Q_R$ 。容易看出， $Q_L$ 和 $Q_R$ 将自动按照 $y$ 坐标排序。当递归调用返回时，我们扫描 $Q$ 表并删除其 $x$ 坐标不在带内的所有点。此时 $Q$ 只含有带中的点，而这些点保证是按照 $y$ 坐标排序的。

434

这种策略保证整个算法是 $O(M\log N)$ 的，因为只执行了 $O(N)$ 的附加工作。

### 10.2.3 选择问题

**选择问题** (selection problem) 要求找出含 $N$ 个元素的集合 $S$ 中的第 $k$ 个最小的元素。我们对找出中间元素的特殊情况特别感兴趣，这种情况发生在 $k = \lceil N/2 \rceil$ 的时候。

在第1章、第6章和第7章我们已经看到过选择问题的几种解法。第7章中的解法用到快速排序的变体并以平均时间 $O(N)$ 运行。事实上，它在Hoare论述快速排序的原始论文中已有描述。

虽然这个算法以线性平均时间运行，但是它有一个 $O(N^2)$ 的最坏情形。通过把元素排序，选择可以容易地以 $O(M\log N)$ 最坏情形时间解决，不过，曾经长期不知道选择是否能够以 $O(N)$ 最坏情形时间完成。7.7.6节概述的快速选择算法在实践中是相当有效的，因此这个问题主要还是理论上的问题。

我们知道，基本的算法是简单递归策略。设 $N$ 大于截止点 (cutoff point)，元素将从截止点开始进行简单的排序， $v$ 是选出的一个元素，叫作枢纽元 (pivot)。其余的元素放在两个集合 $S_1$ 和 $S_2$ 中。 $S_1$ 含有那些不大于 $v$ 的元素，而 $S_2$ 则包含那些不小于 $v$ 的元素。最后，如果 $k \leq |S_1|$ ，那么 $S$ 中的第 $k$ 个最小的元素可以通过递归地计算 $S_1$ 中第 $k$ 个最小的元素而找到。如果 $k = |S_1| + 1$ ，则枢纽元就是第 $k$ 个最小的元素。否则， $S$ 中的第 $k$ 个最小的元素是 $S_2$ 中的第 $(k - |S_1| - 1)$ 个最小元素。这个算法和快速排序之间的主要区别在于，这里要求解的只有一个子问题而不是两个子问题。

为了得到一个线性算法，必须保证子问题只是原问题的一部分，而不仅仅只是比原问题少几个元素。当然，如果我们愿意花费一些时间查找的话，那么总能够找到这样的元素。困难在于我们不能花费太多的时间寻找枢纽元。

对于快速排序，我们看到枢纽元的一种好的选择是选取三个元素并取它们的中项。这就产生这种枢纽元不太坏的期望，但它并不提供一种保证。可以随机选取21个元素，以常数时间将它们排序，用第11个最大的元素作为枢纽元，这可能得到更好的枢纽元。然而，如果这21个元素是21个最大元，那么枢纽元仍然不好。将这种想法扩展，可以使用直到 $O(N/\log N)$ 个元素，用堆排序以 $O(N)$ 总时间将它们排序，从统计的观点看几乎肯定会得到一个好的枢纽元。不过，在最坏情形下，这种方法行不通，因为我们可能选择 $O(N/\log N)$ 个最大的元素，而此时的枢纽元则是第 $[N - O(N/\log N)]$ 个最大的元素，这不是 $N$ 的一个常数部分。

然而，基本想法还是有用的。的确，我们将看到，可以用它来改进快速选择所进行的比较的期望次数。但是，为得到一个好的最坏情形，关键想法是再用一个间接层。我们不是从随机元素的样本中找出中项，而是从中项的样本中找出中项。

435

基本的枢纽元选择算法如下：

- (1) 把 $N$ 个元素分成 $\lfloor N/5 \rfloor$ 组，5个元素一组，忽略（最多4个）多余的元素。
- (2) 找出每组的中项，得到个 $\lfloor N/5 \rfloor$ 中项的表 $M$ 。
- (3) 求出 $M$ 的中项，将其作为枢纽元 $v$ 返回。

我们将用术语**五分化中项的中项** (median-of-median-of-five partitioning) 描述使用上面给出的枢纽元选择法则的快速选择算法。现在证明，“五分化中项的中项”保证每个递归子问题的大小最多是原问题的大约70%。我们还要证明，对于整个选择算法，枢纽元可以足够快地算出，以确保 $O(N)$ 的运行时间。



现在假设 $N$ 可以被5整除，因此不存在多余的元素。再设 $N/5$ 为奇数，这样集 $M$ 就包含奇数个元素。我们将要看到，这将提供某种对称性。于是，为方便起见假设 $N$ 为 $10k+5$ 的形式。还要假设所有的元素都是互异的。实际的算法必须保证能够处理该假设不成立的情况。图10-36指出当 $N=45$ 时如何选出枢纽元。

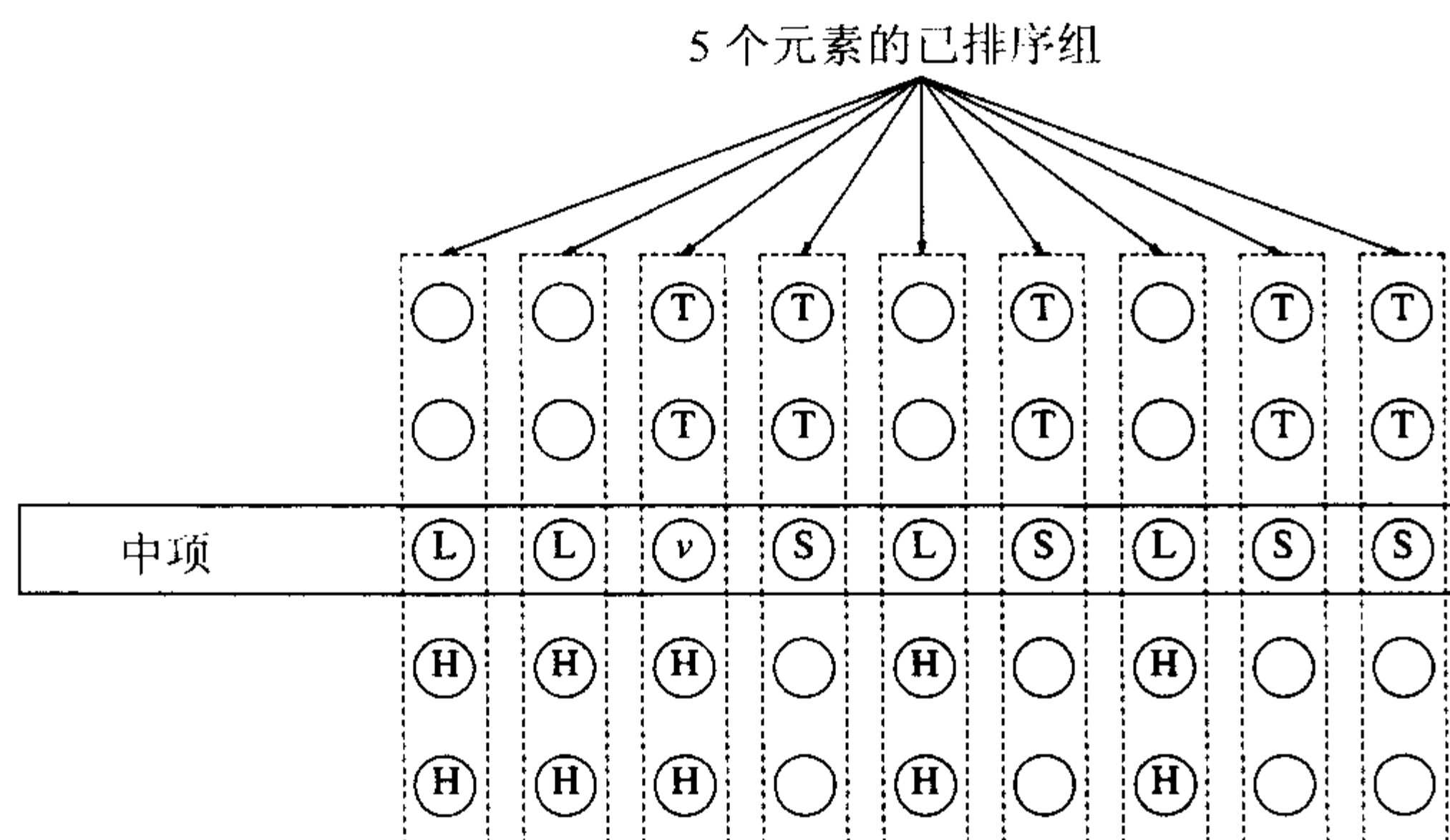


图10-36 枢纽元的选择

在图10-36中， $v$ 代表该算法选出作为枢纽元的元素。由于 $v$ 是9个元素的中项，而我们假设所有元素互异，因此必然存在4个中项大于 $v$ 以及4个中项小于 $v$ 。我们分别用 $L$ 和 $S$ 表示这些中项。考虑具有一个大中项( $L$ 型)的5元素组。该组的中项小于组中的两个元素且大于组中的另两个元素。令 $H$ 代表那些巨型元素，存在一些已知大于大中项的元素。类似地，令 $T$ 代表那些小于小中项的小型元素。存在10个 $H$ 型的元素：具有 $L$ 型中项的每组中有两个， $v$ 所在的组中有两个。类似地，存在10个 $T$ 型元素。

$L$ 型元素或 $H$ 型元素保证大于 $v$ ，而 $S$ 型元素或 $T$ 型元素保证小于 $v$ 。于是在该问题中保证有14个大元素和14个小元素。因此，递归调用最多可以对 $45 - 14 - 1 = 30$ 个元素进行。

让我们把分析推广到对形如 $10k+5$ 的一般的 $N$ 的情形。在这种情况下，存在 $k$ 个 $L$ 型元素和 $k$ 个 $S$ 型元素。存在 $2k+2$ 个 $H$ 型元素，还有 $2k+2$ 个 $T$ 型元素。因此，有 $3k+2$ 个元素保证大于 $v$ 以及 $3k+2$ 个元素保证小于 $v$ 。于是，在这种情况下递归调用最多可以包含 $7k+2 < 0.7N$ 个元素。如果 $N$ 不是 $10k+5$ 的形式，仍可进行类似的论证而不影响基本结果。

剩下的问题是确定得到枢纽元的运行时间的界。有两个基本的步骤。可以以常数时间找到5个元素的中项，例如，不难用8次比较将5个元素排序。必须进行 $\lfloor N/5 \rfloor$ 次这样的运算，因此这一步花费 $O(N)$ 时间。然后必须计算 $\lfloor N/5 \rfloor$ 元素组的中项，明显的做法是将该组排序并返回中间的元素，但这需要花费 $O(\lfloor N/5 \rfloor \log \lfloor N/5 \rfloor) = O(N \log N)$ 的时间，因此不能这么做。解决方法是对这 $\lfloor N/5 \rfloor$ 个元素递归地调用选择算法。

现在对基本算法的描述已经完成。如果想有一个实际的实现方法，那么还有某些细节仍然需要填补。例如，重复元必须要正确地处理，该算法需要截止点足够大以确保递归调用能够进行。由于涉及相当大量的系统开销，而且该算法根本不实用，因此我们将不再描述需要考虑的任何细节。即便如此，该算法从理论的角度来看仍然是一种突破，因为其运行时间在最坏情形下是线性的，这正如下面的定理所述。

**定理10.9** 使用“五分化中项的中项”的快速选择算法的运行时间为 $O(N)$ 。

**证明** 该算法由大小分别为 $0.7N$ 和 $0.2N$ 的两个递归调用以及线性附加工作组成。根据定理10.8，其运行时间是线性的。 ■

### 降低比较的平均次数

分治算法还可以用来降低选择算法所需要的期望的比较次数。下面看一个具体的例子。设有1000个数的集合 $S$ 并且要寻找其中第100个最小的数 $X$ 。选择 $S$ 的子集 $S'$ ，它由100个数组成。我们期望 $X$ 的值在大小上类似于 $S'$ 的第10个最小的数。尤其是 $S'$ 的第5个最小的数几乎可以肯定小于 $X$ ，而 $S'$ 的第15个最小的数几乎可以肯定大于 $X$ 。

更一般地，从 $N$ 个元素选取 $s$ 个元素的样本 $S'$ 。令 $\delta$ 是某个数，后面我们将选择它，使得把该过程所用的平均比较次数最小化。我们找出 $S'$ 中第 $v_1(= ks/N - \delta)$ 个和第 $(v_2 = ks/N + \delta)$ 个最小的元素。几乎可以肯定 $S$ 中的第 $k$ 个最小元素将落在 $v_1$ 和 $v_2$ 之间，因此留给我们的是关于 $2\delta$ 个元素的选择问题。第 $k$ 个最小元素不落在该范围内的概率很低，而我们有大量的工作要做。不过，只要 $s$ 和 $\delta$ 选择得好，根据概率论的定律，可以肯定第二种情形对于整体工作不会有不利的影响。

437

如果进行分析，那么就会发现：若 $s = N^{2/3} \log^{1/3} N$ 和 $\delta = N^{1/3} \log^{2/3} N$ ，则期望的比较次数为 $N + k + O(N^{2/3} \log^{1/3} N)$ ，除低次项外它是最优的。（如果 $k > N/2$ ，那么可以考虑查找第 $N - k$ 个最大元素的对称问题。）

大部分的分析都很容易进行。最后一项代表进行两次选择以确定 $v_1$ 和 $v_2$ 的代价。假设采用合理的“聪明”策略，则划分的平均代价为 $N$ 加上 $v_2$ 在 $S$ 中的期望阶，即 $N + k + O(N\delta/s)$ 。如果第 $k$ 个元素在 $S'$ 中出现，那么结束算法的代价等于对 $S'$ 进行选择代价，即 $O(s)$ 。如果第 $k$ 个最小元素不在 $S'$ 中出现，那么代价就是 $O(N)$ 。然而， $s$ 和 $\delta$ 已经被选取以保证这种情况以非常低的概率 $o(1/N)$ 发生，因此该可能性的期望代价是 $o(1)$ ，当 $N$ 越来越大时它趋向于0。一种精确的计算留作练习10.21。

这个分析表明，找出中项平均大约需要 $1.5N$ 次比较。当然，该算法为计算 $s$ 需要浮点运算，这在一些机器上可能使该算法减慢速度。不过即使是这样，经验表明，若能正确实现，该算法完全比得上第7章中的快速选择实现。

### 10.2.4 一些算术问题的理论改进

本节描述一个分治算法，该算法是将两个 $N$ 位数相乘。前面的计算模型假设乘法是以常数时间完成的，因为相乘的数很小。对于大的数，这个假设不再有效。如果以参加相乘的数的大小来衡量乘法，那么自然的乘法算法花费二次时间，而分治算法则以亚二次（subquadratic）时间运行。我们还要介绍经典的分治算法，它以亚立方时间将两个 $N \times N$ 矩阵相乘。

#### 1. 整数相乘

设想要将两个 $N$ 位数 $X$ 和 $Y$ 相乘。如果 $X$ 和 $Y$ 恰好有一个是负的，那么结果就是负的；否则结果为正数。因此，可以进行这种检查，然后假设 $X, Y \geq 0$ 。几乎每个人在手算乘法时使用的算法都需要 $\Theta(N^2)$ 次操作，这是因为 $X$ 的每一位数字都要被 $Y$ 的每一位数字去乘。

如果 $X = 61\,438\,521$ 而 $Y = 94\,736\,407$ ，那么 $XY = 5\,820\,464\,730\,934\,047$ 。把 $X$ 和 $Y$ 拆成两半，分别由最高几位和最低几位数字组成。此时， $X_L = 6143$ ， $X_R = 8521$ ， $Y_L = 9473$ ， $Y_R = 6407$ 。这样， $X = X_L 10^4 + X_R$ 以及 $Y = Y_L 10^4 + Y_R$ 。由此得到

$$XY = X_L Y_L 10^8 + (X_L Y_R + X_R Y_L) 10^4 + X_R Y_R$$

438

注意，这个方程由4次乘法组成，即 $X_L Y_L$ ， $X_L Y_R$ ， $X_R Y_L$ 和 $X_R Y_R$ ，每一个都是原问题大小的一半（ $N/2$ 位数字）。用 $10^8$ 和 $10^4$ 做乘法实际就是添加一些0，这及其后的几次加法只是添加了 $O(N)$ 附加工作。如果递归地使用该算法进行这4项乘法，在一个适当的基准情形下停止，那么得到递推

$$T(N) = 4T(N/2) + O(N)$$

从定理10.6可以看到,  $T(N) = O(N^2)$ , 因此这根本没有改进这个算法。为了得到一个亚二次的算法, 必须使用少于4次的递归调用。关键的观察结果是

$$X_L Y_R + X_R Y_L = (X_L - X_R)(Y_R - Y_L) + X_L Y_L + X_R Y_R$$

于是, 不用两次乘法来计算 $10^4$ 的系数, 而可以用一次乘法再加上已经完成的两次乘法的结果。图10-37演示了如何只求解3次递推子问题。

| 函数                                  | 值                     | 计算复杂度    |
|-------------------------------------|-----------------------|----------|
| $X_L$                               | 6 143                 | 赋值       |
| $X_R$                               | 8 521                 | 赋值       |
| $Y_L$                               | 9 473                 | 赋值       |
| $Y_R$                               | 6 407                 | 赋值       |
| $D_1 = X_L - X_R$                   | -2 378                | $O(N)$   |
| $D_2 = Y_R - Y_L$                   | -3 066                | $O(N)$   |
| $X_L Y_L$                           | 58 192 639            | $T(N/2)$ |
| $X_R Y_R$                           | 54 594 047            | $T(N/2)$ |
| $D_1 D_2$                           | 7 290 948             | $T(N/2)$ |
| $D_3 = D_1 D_2 + X_L Y_L + X_R Y_R$ | 120 077 634           | $O(N)$   |
| $X_R Y_R$                           | 54 594 047            | 上页已算出    |
| $D_3 10^4$                          | 1 200 776 340 000     | $O(N)$   |
| $X_L Y_L 10^8$                      | 5 819 263 900 000 000 | $O(N)$   |
| $X_L Y_L 10^8 + D_3 10^4 + X_R Y_R$ | 5 820 464 730 934 047 | $O(N)$   |

图10-37 分治算法的执行情况

容易看到现在的递推方程满足

$$T(N) = 3T(N/2) + O(N)$$

从而得到 $T(N) = O(N^{\log_2 3}) = O(N^{1.59})$ 。为完成这个算法, 必须要有一个基准情形, 该情形可以无需递归而解决。

当两个数都是一位数字时, 可以通过查表进行乘法; 若有一个乘数为0, 则返回0。假如在实践中要用这种算法, 那么就要选择对机器最方便的情况作为基准情形。

虽然这种算法比标准的二次算法有更好的渐近性能, 但是却很少使用, 因为对于小的 $N$ 开销大, 而对大的 $N$ 还存在更好的一些算法, 这些算法也广泛利用了分治策略。

## 2. 矩阵乘法

一个基本的数值问题是两个矩阵的乘法。图10-38给出了一个简单的 $O(N^3)$ 算法计算 $C = AB$ , 其中 $A$ 、 $B$ 和 $C$ 均为 $N \times N$ 矩阵。该算法直接来自于矩阵乘法的定义。为了计算 $C_{i,j}$ , 计算 $A$ 的第 $i$ 行和 $B$ 的第 $j$ 列的点乘。按照通常的惯例, 数组下标均从0开始。

长期以来曾认为矩阵乘法是需要工作量 $\Omega(N^3)$ 的。但是, 在20世纪60年代末Strassen指出了如何打破 $\Omega(N^3)$ 的屏障。Strassen算法的基本想法是把每一个矩阵都分成4块, 如图10-39所示。此时容易证明:

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

```

1 /**
2  * Standard matrix multiplication.
3  * Arrays start at 0.
4  * Assumes a and b are square.
5  */
6 matrix<int> operator*( const matrix<int> & a, const matrix<int> & b )
7 {
8     int n = a.numrows( );
9     matrix<int> c( n, n );
10
11     int i;
12     for( i = 0; i < n; i++ )    // Initialization
13         for( int j = 0; j < n; j++ )
14             c[ i ][ j ] = 0;
15
16     for( i = 0; i < n; i++ )
17         for( int j = 0; j < n; j++ )
18             for( int k = 0; k < n; k++ )
19                 c[ i ][ j ] += a[ i ][ k ] * b[ k ][ j ];
20
21     return c;
22 }

```

图10-38 简单的 $O(N^3)$ 矩阵乘法

440

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

图10-39 把 $\mathbf{AB} = \mathbf{C}$ 分解成4块乘法

作为一个例子，为了进行乘法 $\mathbf{AB}$ ：

$$\mathbf{AB} = \begin{bmatrix} 3 & 4 & 1 & 6 \\ 1 & 2 & 5 & 7 \\ 5 & 1 & 2 & 9 \\ 4 & 3 & 5 & 6 \end{bmatrix} \begin{bmatrix} 5 & 6 & 9 & 3 \\ 4 & 5 & 3 & 1 \\ 1 & 1 & 8 & 4 \\ 3 & 1 & 4 & 1 \end{bmatrix}$$

我们定义下列8个 $N/2 \times N/2$ 阶矩阵：

$$\begin{aligned} A_{1,1} &= \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix} & A_{1,2} &= \begin{bmatrix} 1 & 6 \\ 5 & 7 \end{bmatrix} & B_{1,1} &= \begin{bmatrix} 5 & 6 \\ 4 & 5 \end{bmatrix} & B_{1,2} &= \begin{bmatrix} 9 & 3 \\ 3 & 1 \end{bmatrix} \\ A_{2,1} &= \begin{bmatrix} 5 & 1 \\ 4 & 3 \end{bmatrix} & A_{2,2} &= \begin{bmatrix} 2 & 9 \\ 5 & 6 \end{bmatrix} & B_{2,1} &= \begin{bmatrix} 1 & 1 \\ 3 & 1 \end{bmatrix} & B_{2,2} &= \begin{bmatrix} 8 & 4 \\ 4 & 1 \end{bmatrix} \end{aligned}$$

此时，可以进行8个 $N/2 \times N/2$ 阶矩阵的乘法和4个 $N/2 \times N/2$ 阶矩阵的加法。这些加法花费 $O(N^2)$ 时间。如果递归地进行矩阵乘法，那么运行时间满足

$$T(N) = 8T(N/2) + O(N^2)$$

从定理10.6可以看到 $T(N) = O(N^3)$ ，因此这没有对算法做出改进。如同在整数乘法中看到的，必须把子问题的个数简化到8个以下。Strassen使用了类似于整数乘法分治算法的策略，并指出如何仔细地安排计算只使用7次递归调用。这7个乘法是

$$M_1 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$



$$M_2 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$

$$M_3 = (A_{1,1} - A_{2,1})(B_{1,1} + B_{1,2})$$

$$M_4 = (A_{1,1} + A_{1,2})B_{2,2}$$

$$M_5 = A_{1,1}(B_{1,2} - B_{2,2})$$

$$M_6 = A_{2,2}(B_{2,1} - B_{1,1})$$

$$M_7 = (A_{2,1} + A_{2,2})B_{1,1}$$

441

一旦执行这些乘法，则最后答案可以通过下列8次加法得到：

$$C_{1,1} = M_1 + M_2 - M_4 + M_6$$

$$C_{1,2} = M_4 + M_5$$

$$C_{2,1} = M_6 + M_7$$

$$C_{2,2} = M_2 - M_3 + M_5 - M_7$$

可以直接验证，这种机敏的安排产生了期望的效果。现在运行时间满足递推关系

$$T(N) = 7T(N/2) + O(N^2)$$

这个递推关系的解为  $T(N) = O(N^{\log_2 7}) = O(N^{2.81})$ 。

如往常一样，有些细节需要考虑，如当  $N$  不是 2 的幂时的情况，不过还是有些根本性的小缺憾。Strassen 算法在  $N$  不够大时不如矩阵直接乘法，它也不能推广到矩阵是稀疏矩阵（即含有许多 0 元素）的情况，而且还不容易并行化。当用浮点项运算时，在数值上它不如经典的算法稳定。因此，它只有有限的适用性。然而，它却代表着重要的理论上的里程碑，而且证明了：在计算机科学中像在许多其他领域一样，即使一个问题看似具有固有的复杂性，但在被证明以前却始终不可定论。

## 10.3 动态规划

在前一节，我们看到一个可以被数学上递归表示的问题也可以表示成递归算法，在许多情形下对朴素的穷举搜索得到显著的性能改进。

任何数学递推公式都可以直接翻译成递归算法，但是基本现实是编译器常常不能正确地对待递归算法，结果产生低效的程序。当怀疑可能是这种情况时，必须再给编译器提供一些帮助，将递归算法重新写成非递归算法，让后者把那些子问题的答案系统地记录在一个表（table）内。利用这种方法的一种技巧称为动态规划（dynamic programming）。

### 10.3.1 用表代替递归

在第2章我们看到，计算斐波那契数的自然递归程序是非常低效的。前面曾介绍过，图10-40所示的程序的运行时间  $T(N)$  满足  $T(N) \geq T(N-1) + T(N-2)$ 。由于  $T(N)$  满足与斐波那契数相同的递推关系并具有相同的初始条件，因此，事实上  $T(N)$  是以与斐波那契数相同的速度在增长，从而是指数级的。

另一方面，由于计算  $F_N$  所需要的只是  $F_{N-1}$  和  $F_{N-2}$ ，因此只需要记录最近算出的两个斐波那契数。这产生图10-41中的  $O(N)$  算法。

递归算法如此慢的原因在于算法用于模仿递推。为了计算  $F_N$ ，存在一个对  $F_{N-1}$  和  $F_{N-2}$  的调用。

442

然而，由于  $F_{N-1}$  递归地对  $F_{N-2}$  和  $F_{N-3}$  进行调用，因此存在两个单独的计算  $F_{N-2}$  的调用。如果试探整

个算法，那么可以发现， $F_{N-3}$ 被计算了3次， $F_{N-4}$ 计算了5次，而 $F_{N-5}$ 则是8次，等等。如图10-42所示，冗余计算的增长是爆炸性的。如果编译器的递归模拟算法能够保留一个预先算出的值的表而对已经解过的子问题不再进行递归调用，那么这种指数式的爆炸增长就可以避免。这就是图10-41中的程序如此有效的原因。

443

```

1  /**
2   * Compute Fibonacci numbers as described in Chapter 1.
3   */
4  int fib( int n )
5  {
6      if( n <= 1 )
7          return 1;
8      else
9          return fib( n - 1 ) + fib( n - 2 );
10 }

```

图10-40 计算斐波那契数的低效算法

```

1  /**
2   * Compute Fibonacci numbers as described in Chapter 1.
3   */
4  int fibonacci( int n )
5  {
6      if( n <= 1 )
7          return 1;
8
9      int last = 1;
10     int nextToLast = 1;
11     int answer = 1;
12     for( int i = 2; i <= n; i++ )
13     {
14         answer = last + nextToLast;
15         nextToLast = last;
16         last = answer;
17     }
18     return answer;
19 }

```

图10-41 计算斐波那契数的线性算法

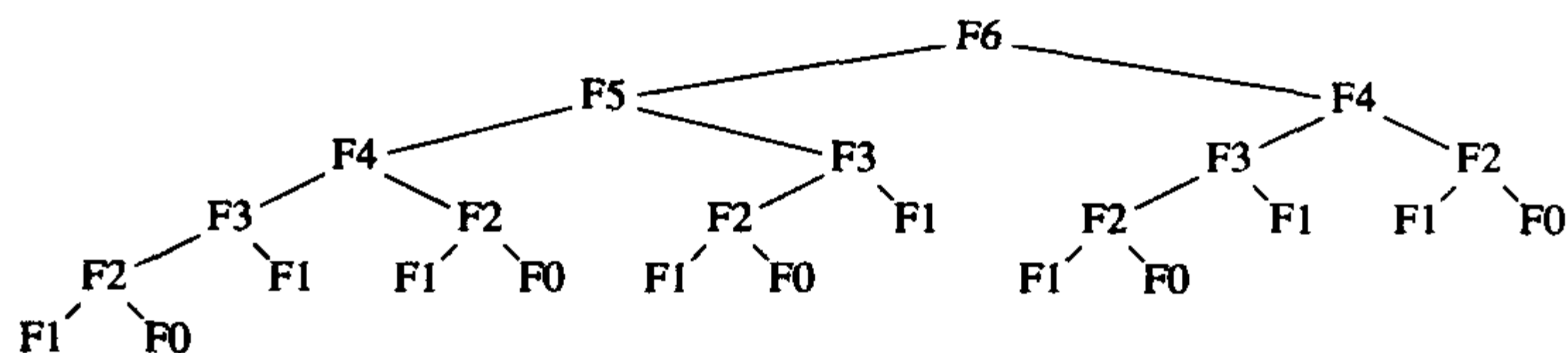


图10-42 跟踪斐波那契数的递归计算

作为第二个例子，我们看看第7章中如何求解递推关系 $C(N)=(2/N)\sum_{i=0}^{N-1}C(i)+N$ ，其中 $C(0)=1$ 。假设我们想要检查所得到的解是否在数值上是正确的，此时可以编写图10-43中的简单程序来计算这个递归问题。

这里，递归调用又做了重复性的工作。在这种情况下，运行时间 $T(N)$ 满足 $T(N)=\sum_{i=0}^{N-1}T(i)+N$ ，因为，如图10-44所示，对于从0到 $N-1$ 的每一个值都有一个（直接的）递归调用，外加 $O(N)$ 的附加工作。（图10-44所示的树我们还在哪里看到过？）对 $T(N)$ 求解可以发现，它的增长是指数级的。

通过使用表，得到图10-45中的程序，这个程序避免了冗余的递归调用而以 $O(N^2)$ 运行。它并不是一个完美的程序；作为练习，可对它做些简单修改，把它的运行时间减到 $O(N)$ 。

```

1 double eval( int n )
2 {
3     if( n == 0 )
4         return 1.0;
5     else
6     {
7         double sum = 0.0;
8         for( int i = 0; i < n; i++ )
9             sum += eval( i );
10        return 2.0 * sum / n + n;
11    }
12 }

```

图10-43 计算 $C(N) = 2/N \sum_{i=0}^{N-1} C(i) + N$ 的值的递归函数

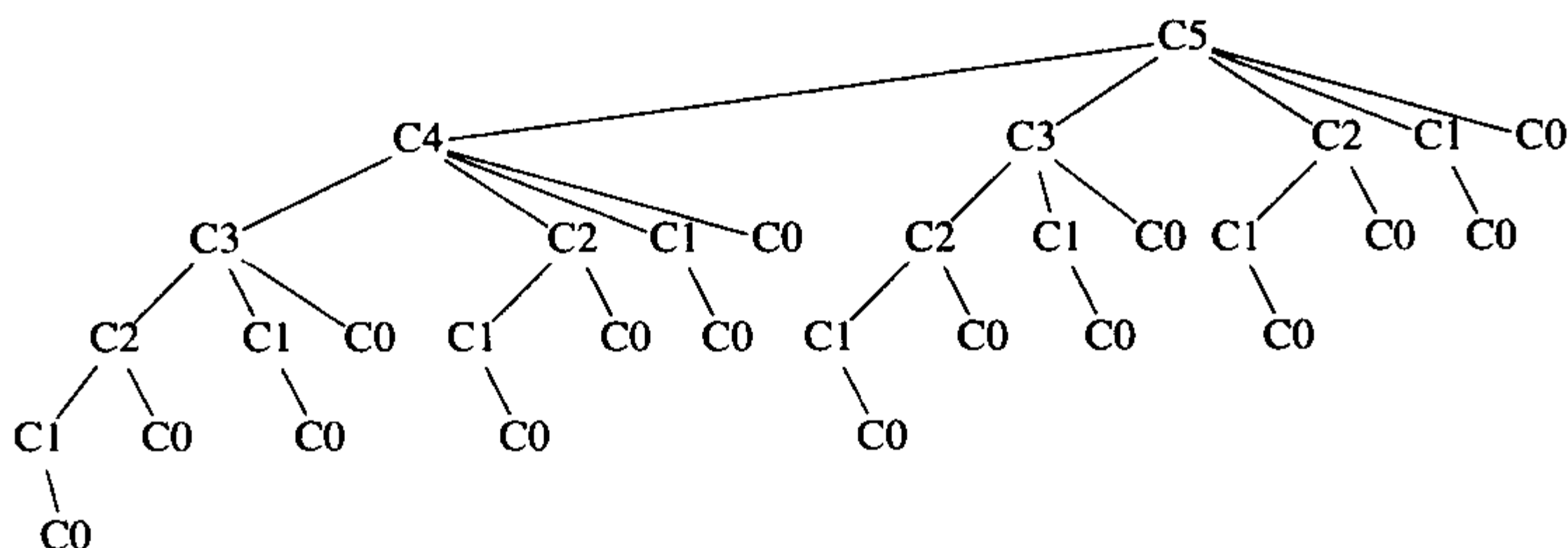


图10-44 跟踪方法eval中的递归计算

```

1 double eval( int n )
2 {
3     vector<double> c( n + 1 );
4
5     c[ 0 ] = 1.0;
6     for( int i = 1; i <= n; i++ )
7     {
8         double sum = 0.0;
9         for( int j = 0; j < i; j++ )
10            sum += c[ j ];
11        c[ i ] = 2.0 * sum / i + i;
12    }
13
14    return c[ n ];
15 }

```

图10-45 使用表来计算 $C(N) = 2/N \sum_{i=0}^{N-1} C(i) + N$ 的值

### 10.3.2 矩阵乘法的顺序安排

设给定4个矩阵A、B、C和D，A的大小=  $50 \times 10$ ，B的大小=  $10 \times 40$ ，C的大小=  $40 \times 30$ ，D的大小=  $30 \times 5$ 。虽然矩阵乘法运算是不可交换的，但是它是可结合的，这就意味着矩阵的乘积ABCD可以以任意顺序添加括号然后再计算其值。将两个阶数分别为 $p \times q$ 和 $q \times r$ 的矩阵相乘明显的方法是使用 $pqr$ 次标量乘法。（由于使用诸如Strassen算法这样的理论上优越的算法并没有明显地改变要考虑的问题，因此我们将采用上面的性能界）。那么，计算ABCD需要执行的三个矩阵乘法的最好方式是什么？

在4个矩阵的情况下,通过穷举搜索求解这个问题是很简单的,因为只有5种方式来给乘法排顺序。我们对每种情况计算如下:

- $(A((BC)D))$ : 计算 $BC$ 需要 $10 \times 40 \times 30 = 12\,000$ 次乘法。计算 $(BC)D$ 的值需要12 000次乘法计算 $BC$ ,外加 $10 \times 30 \times 5 = 1500$ 次乘法,合计13 500次乘法。求 $(A((BC)D))$ 的值需要13 500次乘法计 $(BC)D$ ,外加 $50 \times 10 \times 5 = 2500$ 次乘法,总计16 000次乘法。
- $(A(B(CD)))$ : 计算 $CD$ 需要 $40 \times 30 \times 5 = 6000$ 次乘法。计算 $B(CD)$ 的值需要6000次乘法计算 $CD$ ,外加 $10 \times 40 \times 5 = 2000$ 次乘法,合计8000次乘法。求 $(A(B(CD)))$ 的值需要8000次乘法计算 $B(CD)$ ,外加 $50 \times 10 \times 5 = 2500$ 次乘法,总计10 500次乘法。
- $((AB)(CD))$ : 计算 $CD$ 需要 $40 \times 30 \times 5 = 6000$ 次乘法。计算 $AB$ 需要 $50 \times 10 \times 40 = 20\,000$ 次乘法。求 $((AB)(CD))$ 的值需要6000次乘法计算 $CD$ ,20 000次乘法计算 $AB$ ,外加 $50 \times 40 \times 5 = 10\,000$ 次乘法,总计36 000次乘法。
- $((AB)C)D$ : 计算 $AB$ 需要 $50 \times 10 \times 40 = 20\,000$ 次乘法。计算 $(AB)C$ 的值需要20 000次乘法计算 $AB$ ,外加 $50 \times 40 \times 30 = 60\,000$ 次乘法,合计80 000次乘法。求 $((AB)C)D$ 的值需要80 000次乘法计算 $(AB)C$ ,外加 $50 \times 30 \times 5 = 7500$ 次乘法,总计87 500次乘法。
- $((A(BC))D)$ : 计算 $BC$ 需要 $10 \times 40 \times 30 = 12\,000$ 次乘法。计算 $A(BC)$ 的值需要12 000次乘法计算 $BC$ ,外加 $50 \times 10 \times 30 = 15\,000$ 次乘法,合计27 000次乘法。求 $((A(BC))D)$ 的值需要27 000次乘法计算 $A(BC)$ ,外加 $50 \times 30 \times 5 = 7500$ 次乘法,总计34 500次乘法。

445

上面的计算表明,最好的排列顺序方法大约只用了最坏的排列顺序方法的九分之一的乘法次数。因此,进行一些计算来确定最优顺序还是值得的。但是,一些明显的贪心算法似乎都用不上,而且可能的顺序的个数增长很快。假设定义 $T(N)$ 是顺序的个数,此时, $T(1) = T(2) = 1$ , $T(3) = 2$ ,而 $T(4) = 5$ ,正如前面所看到的。一般地,

$$T(N) = \sum_{i=1}^{N-1} T(i)T(N-i)$$

为此,设矩阵为 $A_1, A_2, \dots, A_N$ ,且最后进行的乘法是 $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_N)$ 。此时,有 $T(i)$ 种方法计算 $(A_1 A_2 \cdots A_i)$ 且有 $T(N-i)$ 种方法计算 $(A_{i+1} A_{i+2} \cdots A_N)$ 。因此,对于每个可能的 $i$ ,存在 $T(i)T(N-i)$ 种方法计算 $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_N)$ 。

这个递推式的解是著名的Catalan数,该数呈指数增长。因此,对于大的 $N$ ,穷举搜索所有可能的排列顺序的方法是不可行的。然而,这种计数方法为一种解法提供了基础,该解法基本上是优于指数的。对于 $1 \leq i \leq N$ ,令 $c_i$ 是矩阵 $A_i$ 的列数。于是 $A_i$ 有 $c_{i-1}$ 行,否则矩阵乘法是无法进行的。定义 $c_0$ 为第一个矩阵 $A_1$ 的行数。

设 $m_{Left,Right}$ 是进行矩阵乘法 $A_{Left} A_{Left+1} \cdots A_{Right-1} A_{Right}$ 所需要的乘法次数。为一致起见,设 $m_{Left,Left} = 0$ 。设最后的乘法是 $(A_{Left} \cdots A_i)(A_{i+1} \cdots A_{Right})$ ,其中 $Left \leq i < Right$ 。此时所用的乘法次数为 $m_{Left,i} + m_{i+1,Right} + c_{Left-1} c_i c_{Right}$ ,这三项分别代表计算 $(A_{Left} \cdots A_i)$ 、 $(A_{i+1} \cdots A_{Right})$ 以及它们的乘积所需要的乘法次数。

446

如果定义 $M_{Left,Right}$ 为最优排列顺序下所需要的乘法次数,那么,若 $Left < Right$ ,则

$$M_{Left,Right} = \min_{Left \leq i < Right} \{M_{Left,i} + M_{i+1,Right} + c_{Left-1} c_i c_{Right}\}$$

这个方程意味着,如果有乘法 $A_{Left} \cdots A_{Right}$ 的最优的乘法排列顺序,那么子问题 $A_{Left} \cdots A_i$ 和 $A_{i+1} \cdots A_{Right}$ 就不能次最优地执行。这是很清楚的,因为,否则可以通过用最优的计算代替次最优计算而改进整个结果。

这个公式可以直接翻译成递归程序,不过,正如前面所看到的,这样的程序将是明显低效的。



然而, 由于大约只有 $M_{Left, Right}$ 的 $N^2/2$ 个值需要计算, 因此显然可以用一个表来存放这些值。进一步的考察表明, 如果 $Right - Left = k$ , 那么只有在 $M_{Left, Right}$ 的计算所需要的那些值 $M_{x,y}$ 满足 $x - y < k$ 。这告诉我们计算这个表所需要使用的顺序。

如果除最后答案 $M_{1,N}$ 外还想要显示实际的乘法顺序, 那么可以使用第9章中的最短路径算法的思路。无论何时改变 $M_{Left, Right}$ , 都要记录 $i$ 的值, 这个值是重要的。由此得到图10-46所示的简单程序。

```

1  /**
2   * Compute optimal ordering of matrix multiplication.
3   * c contains the number of columns for each of the n matrices.
4   * c[ 0 ] is the number of rows in matrix 1.
5   * The minimum number of multiplications is left in m[ 1 ][ n ].
6   * Actual ordering is computed via another procedure using lastChange.
7   * m and lastChange are indexed starting at 1, instead of 0.
8   * Note: Entries below main diagonals of m and lastChange
9   * are meaningless and uninitialized.
10  */
11 void optMatrix( const vector<int> & c,
12                matrix<long> & m, matrix<int> & lastChange )
13 {
14     int n = c.size( ) - 1;
15
16     for( int left = 1; left <= n; left++ )
17         m[ left ][ left ] = 0;
18     for( int k = 1; k < n; k++ ) // k is right - left
19         for( int left = 1; left <= n - k; left++ )
20             {
21                 // For each position
22                 int right = left + k;
23                 m[ left ][ right ] = INFINITY;
24                 for( int i = left; i < right; i++ )
25                     {
26                         long thisCost = m[ left ][ i ] + m[ i + 1 ][ right ]
27   + c[ left - 1 ] * c[ i ] * c[ right ];
28                         if( thisCost < m[ left ][ right ] ) // Update min
29                             {
30                                 m[ left ][ right ] = thisCost;
31                                 lastChange[ left ][ right ] = i;
32                             }
33                     }
34             }
35 }

```

图10-46 找出矩阵乘法最优顺序的程序

虽然本章的重点不是编程, 但是还是要提醒大家注意, 许多编程人员倾向于把变量名称减缩成一个字母, 这并没有什么好处。可这里 $c$ 、 $i$ 和 $k$ 却是作为单字母变量使用的, 这是因为它们与我们描述算法所使用的名字是一致的, 是非常数学化的。不过, 一般最好避免使用字母“1”作为变量名, 因为“1”(字母)非常像“1”(数字), 如果你犯了抄写错误, 那么可能会陷入非常困难的调试之中。

回到算法问题上来。这个程序包含三重嵌套循环, 容易看出它以 $O(N^3)$ 时间运行。参考文献描述了一个更快的算法, 但由于执行具体矩阵乘法的时间仍然可能比计算最优顺序的乘法的时间多得多, 因此这个算法还是相当实用的。

10.3.3 最优二叉查找树

第二个动态规划的例子考虑下列输入：给定一组单词 $w_1, w_2, \dots, w_N$  和它们出现的固定概率 $p_1, p_2, \dots, p_N$ 。问题是要以某种方法在一棵二叉查找树中安放这些单词使得总的期望访问时间最小。在一棵二叉查找树中，访问深度 $d$ 处的一个元素所需要的比较次数是 $d + 1$ ，因此如果 $w_i$ 被放在深度 $d_i$ 上，那么就要将 $\sum_{i=1}^N p_i(1 + d_i)$  极小化。

作为一个例子，图10-47表示某个上下文中的7个单词以及它们出现的概率。图10-48显示了3棵可能的二叉查找树。它们的查找开销如图10-49所示。

| 单词  | 概率   |
|-----|------|
| a   | 0.22 |
| am  | 0.18 |
| and | 0.20 |
| egg | 0.05 |
| if  | 0.25 |
| the | 0.02 |
| two | 0.08 |

图10-47 最优二叉查找树问题的样本输入

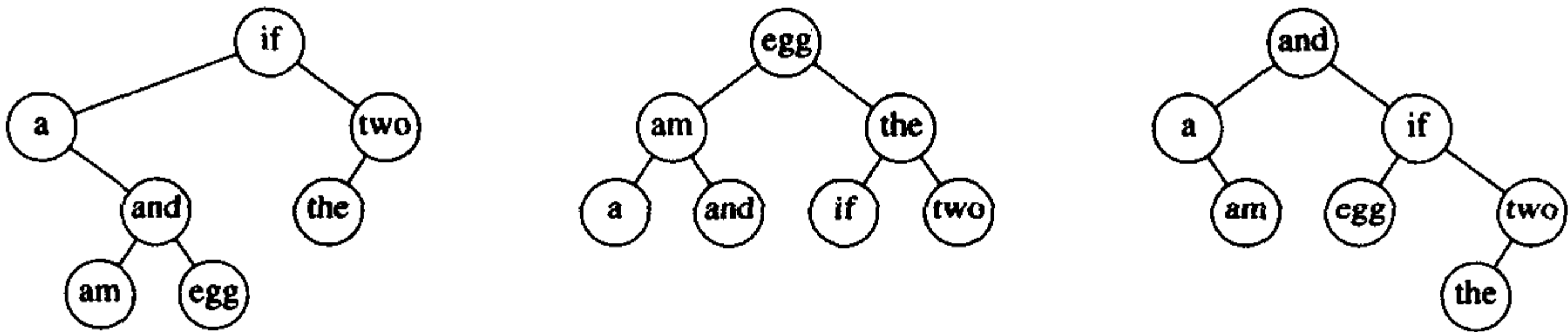


图10-48 对于上表中数据的3棵可能的二叉查找树

| 输入          |             | 第一棵树       |      | 第二棵树       |      | 第三棵树       |      |
|-------------|-------------|------------|------|------------|------|------------|------|
| 单词<br>$w_i$ | 概率<br>$p_i$ | 访问开销<br>一次 | 序列   | 访问开销<br>一次 | 序列   | 访问开销<br>一次 | 序列   |
| a           | 0.22        | 2          | 0.44 | 3          | 0.66 | 2          | 0.44 |
| am          | 0.18        | 4          | 0.72 | 2          | 0.36 | 3          | 0.54 |
| and         | 0.20        | 3          | 0.60 | 3          | 0.60 | 1          | 0.20 |
| egg         | 0.05        | 4          | 0.20 | 1          | 0.05 | 3          | 0.15 |
| if          | 0.25        | 1          | 0.25 | 3          | 0.75 | 2          | 0.50 |
| the         | 0.02        | 3          | 0.06 | 2          | 0.04 | 4          | 0.08 |
| two         | 0.08        | 2          | 0.16 | 3          | 0.24 | 3          | 0.24 |
| 总计          | 1.00        |            | 2.43 |            | 2.70 |            | 2.15 |

图10-49 3棵二叉查找树的比较

第一棵树是使用贪心方法形成的，访问概率最高的单词被放在根结点处，然后左右子树递归形成。第二棵树是理想平衡查找树。这两棵树都不是最优的，由第三棵树可以证实。由此看到，这两个明显的解法都是不可取的。

乍看有些奇怪，因为问题看起来很像是构造赫夫曼编码树，正如我们已经看到的，它能够用贪心算法求解。构造一棵最优二叉查找树更困难，因为数据不只限于出现在树叶上，树还必须满足二叉查找树的性质。

448  
449

动态规划解由两个观察结论得到。再次假设我们想要把(已排序的)一些单词  $w_{Left}, w_{Left+1}, \dots, w_{Right-1}, w_{Right}$  放到一棵二叉查找树中。设最优二叉查找树以  $w_i$  作为根, 其中  $Left \leq i \leq Right$ 。此时左子树必须包含  $w_{Left}, \dots, w_{i-1}$ , 而右子树必须包含  $w_{i+1}, \dots, w_{Right}$  (根据二叉查找树的性质)。再有, 这两棵子树还必须是最优的, 否则它们可以用最优子树代替, 这将给出关于  $w_{Left}, \dots, w_{Right}$  更好的解。这样, 可以为最优二叉查找树的开销  $C_{Left, Right}$  编写一个公式。图10-50可能是有用的。

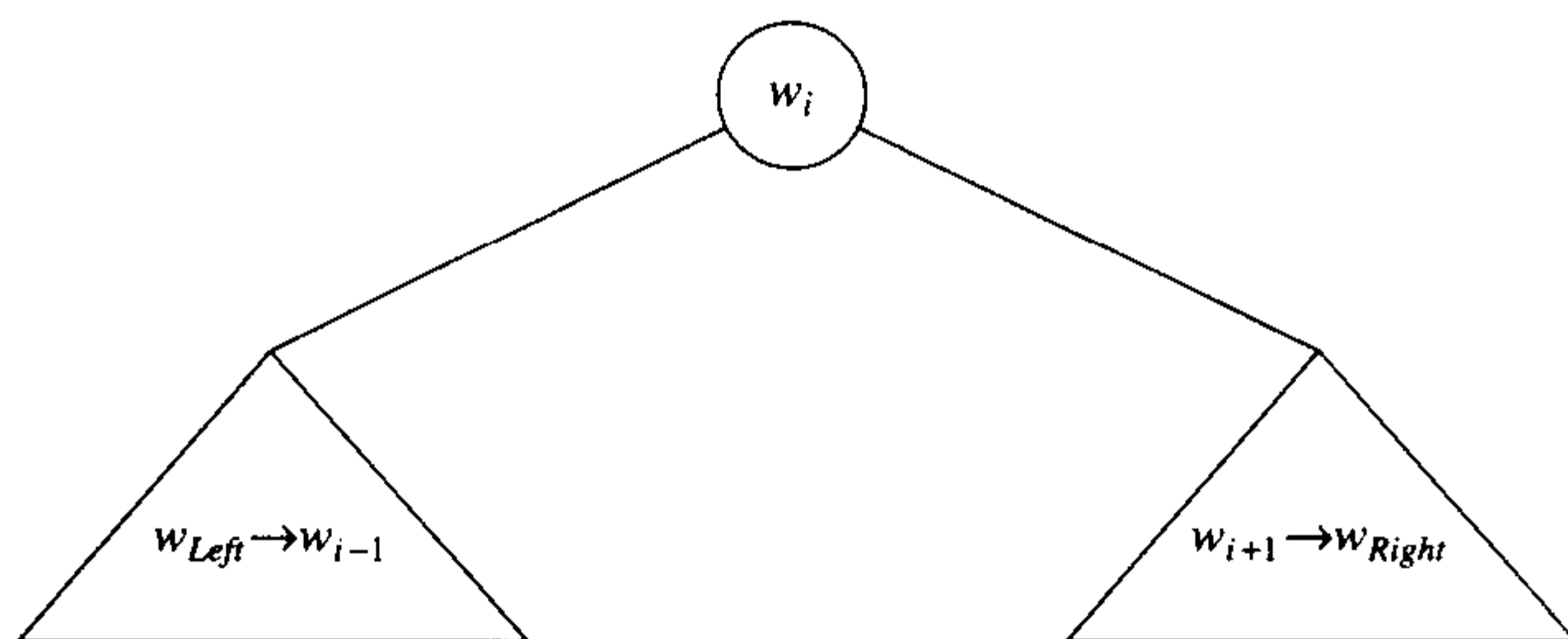


图10-50 最优二叉查找树的构造

如果  $Left > Right$ , 那么树的开销是0; 这就是null情形, 对于二叉查找树总有这种情形。否则, 根花费  $p_i$ 。左子树的开销相对于它的根为  $C_{Left, i-1}$ , 右子树相对于它的根的开销为  $C_{i+1, Right}$ 。如图10-50所示, 这两棵子树的每个结点从  $w_i$  开始都比从它们对应的根开始深一层, 因此, 必须加上  $\sum_{j=Left}^{i-1} p_j$  和  $\sum_{j=i+1}^{Right} p_j$ 。于是得到公式

$$C_{Left, Right} = \min_{Left \leq i \leq Right} \left\{ p_i + C_{Left, i-1} + C_{i+1, Right} + \sum_{j=Left}^{i-1} p_j + \sum_{j=i+1}^{Right} p_j \right\}$$

$$= \min_{Left \leq i \leq Right} \left\{ C_{Left, i-1} + C_{i+1, Right} + \sum_{j=Left}^{Right} p_j \right\}$$

从这个公式可以直接编写一个程序来计算最优二叉查找树的开销。像通常一样, 具体的查找树可以通过存储使  $C_{Left, Right}$  最小化的  $i$  值而保留下来。标准的递归例程可以用来显示具体的树。

图10-51显示了由这一算法产生的表。对于单词的每个子区域, 最优二叉查找树的开销和根都被保留。最底部的项计算输入的全部单词集合的最优二叉查找树。最优树是图10-48中所示的第三棵树。

|      | Left=1               | Left=2                | Left=3                | Left=4                | Left=5              | Left=6                | Left=7                |
|------|----------------------|-----------------------|-----------------------|-----------------------|---------------------|-----------------------|-----------------------|
| 迭代=1 | a..a<br>.22   a      | am..am<br>.18   am    | and..and<br>.20   and | egg..egg<br>.05   egg | if..if<br>.25   if  | the..the<br>.02   the | two..two<br>.08   two |
| 迭代=2 | a..am<br>.58   a     | am..and<br>.56   and  | and..egg<br>.30   and | egg..if<br>.35   if   | if..the<br>.29   if | the..two<br>.12   two |                       |
| 迭代=3 | a..and<br>1.02   am  | am..egg<br>.66   and  | and..if<br>.80   if   | egg..the<br>.39   if  | if..two<br>.47   if |                       |                       |
| 迭代=4 | a..egg<br>1.17   am  | am..if<br>1.21   and  | and..the<br>.84   if  | egg..two<br>.57   if  |                     |                       |                       |
| 迭代=5 | a..if<br>1.83   and  | am..the<br>1.27   and | and..two<br>1.02   if |                       |                     |                       |                       |
| 迭代=6 | a..the<br>1.89   and | am..two<br>1.53   and |                       |                       |                     |                       |                       |
| 迭代=7 | a..two<br>2.15   and |                       |                       |                       |                     |                       |                       |

图10-51 对于样本输入的最优二叉查找树的计算

对于特定子区域am..if的最优二叉查找树的精确计算如图10-52所示。它是通过计算在根处放置am、and、egg和if所得的最小价值树而得到的。例如，and被放在根处时，左子树包含am..am (通过前面的计算，价值为0.18)，右子树包含egg..if (价值为0.35)，而 $p_{am} + p_{and} + p_{egg} + p_{if} = 0.68$ ，总价值为1.21。450

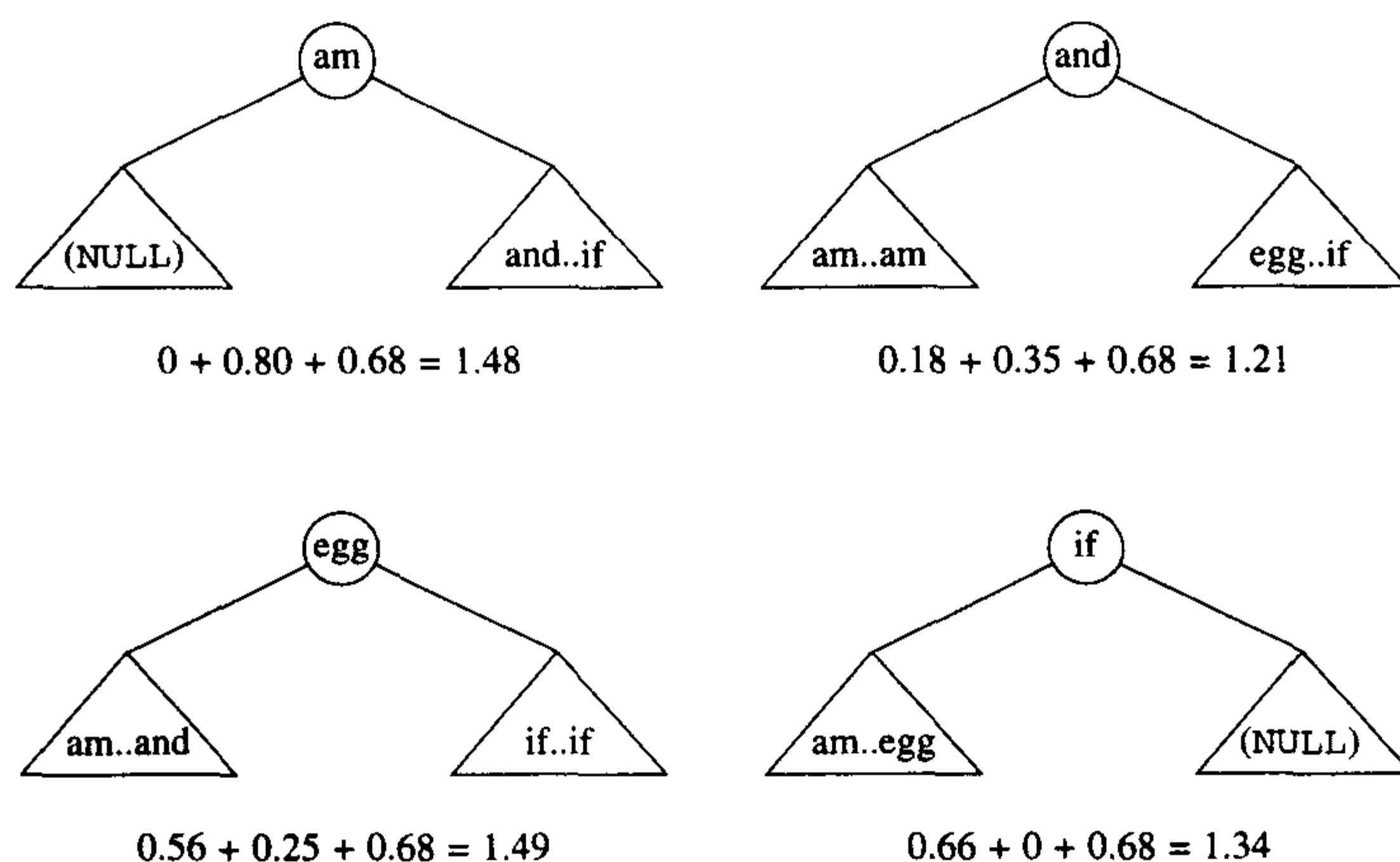


图10-52 对am..if的表项 (1.21, and) 的计算

这个算法的运行时间是 $O(N^3)$ ，因为它实现的时候得到一个三重循环。对于这个问题的一种 $O(N^2)$ 算法在练习中进行了概述。

### 10.3.4 所有点对最短路径

第三个也是最后一个动态规划应用是计算有向图 $G = (V, E)$ 中每一点对间加权最短路径的一个算法。在第9章介绍过单源最短路径问题的一个算法，该算法找出从任意顶点 $s$ 到其他顶点的最短路径。这个 (Dijkstra) 算法对稠密图以 $O(|V|^2)$ 时间运行，但是实际上对稀疏图更快。我们将给出一个短小的算法解决对稠密图的所有点对的问题。这个算法的运行时间为 $O(|V|^3)$ ，它不是对Dijkstra算法 $|V|$ 次迭代的一种渐近改进，但对非常稠密的图可能更快，原因是它的循环更紧凑。如果存在一些负的边值但没有负值回路，那么这个算法也能正确运行；而Dijkstra算法此时是无效的。

下面回顾一下Dijkstra算法的一些重要细节 (读者可以复习9.3节)。Dijkstra算法从顶点 $s$ 开始并分阶段工作。图中的每个顶点最终都要被选作中间顶点。如果当前所选的顶点是 $v$ ，那么对于每个 $w \in V$ ，置 $d_w = \min(d_w, d_v + c_{v,w})$ 。这个公式是说，(从 $s$ )到 $w$ 的最佳距离或者是前面知道的从 $s$ 到 $w$ 的距离，或者是从 $s$  (最优地)到 $v$ 然后再直接从 $v$ 到 $w$ 的结果。451

Dijkstra算法提供了动态规划算法的想法：依序选择这些顶点。将 $D_{k,i,j}$ 定义为从 $v_i$ 到 $v_j$ 只使用 $v_1, v_2, \dots, v_k$ 作为中间顶点的最短路径的权。根据这个定义， $D_{0,i,j} = c_{i,j}$ ，其中若 $(v_i, v_j)$ 不是该图的边则 $c_{i,j}$ 是 $\infty$ 。再有，根据定义， $D_{|V|,i,j}$ 是图中从 $v_i$ 到 $v_j$ 的最短路径。

如图10-53所示，当 $k > 0$ 时可以给 $D_{k,i,j}$ 写一个简单公式。从 $v_i$ 到 $v_j$ 只使用 $v_1, v_2, \dots, v_k$ 作为中间顶点的最短路径或者是根本不使用 $v_k$ 作为中间顶点的最短路径，或者是由两条路径 $v_i \rightarrow v_k$ 和 $v_k \rightarrow v_j$ 合并而成的最短路径，其中每条路径只使用前 $k-1$ 个顶点作为中间顶点。于是有下面的公式：



$$D_{k,i,j} = \min\{D_{k-1,i,j}, D_{k-1,i,k} + D_{k-1,k,j}\}$$

时间需求还是 $O(|V|^3)$ 。与前面的两个动态规划例子不同，这个时间界实际上尚未用其他的方法降低。

```

1  /**
2   * Compute all-shortest paths.
3   * a contains the adjacency matrix with
4   * a[ i ][ i ] presumed to be zero.
5   * d contains the values of the shortest path.
6   * Vertices are numbered starting at 0; all arrays
7   * have equal dimension. A negative cycle exists if
8   * d[ i ][ i ] is set to a negative value.
9   * Actual path can be computed using path[ ][ ].
10  * NOT_A_VERTEX is -1
11  */
12 void allPairs( const matrix<int> & a,
13               matrix<int> & d, matrix<int> & path )
14 {
15     int n = a.numrows( );
16
17     // Initialize d and path
18     for( int i = 0; i < n; i++ )
19         for( int j = 0; j < n; j++ )
20         {
21             d[ i ][ j ] = a[ i ][ j ];
22             path[ i ][ j ] = NOT_A_VERTEX;
23         }
24
25     for( int k = 0; k < n; k++ )
26         // Consider each vertex as an intermediate
27         for( int i = 0; i < n; i++ )
28             for( int j = 0; j < n; j++ )
29                 if( d[ i ][ k ] + d[ k ][ j ] < d[ i ][ j ] )
30                 {
31                     // Update shortest path
32                     d[ i ][ j ] = d[ i ][ k ] + d[ k ][ j ];
33                     path[ i ][ j ] = k;
34                 }
35 }

```

图10-53 所有点对最短路径

因为第 $k$ 阶段只依赖于第 $k-1$ 阶段，所以看来只有两个 $|V| \times |V|$ 矩阵需要保存。然而，在以 $k$ 开始或结束的路径上以 $k$ 作为中间顶点对结果没有改进，除非存在一个负值回路。因此只有一个矩阵是必需的，因为 $D_{k-1,i,k} = D_{k,i,k}$ 和 $D_{k-1,k,j} = D_{k,k,j}$ ，这意味着右边的项都不改变值且都不需要存储。这个观察结果导致图10-53中的简单程序，为与C++的约定一致，该程序将顶点从0开始编号。

**452** 在完全图中，每一对顶点（在两个方向上）都是连通的，该算法几乎肯定要比Dijkstra算法的 $|V|$ 次迭代快，因为这里的循环非常紧凑。第18行到第22行可以并行执行，第27行到第34行也可并行执行。因此，这个算法看起来很适合并行计算。

动态规划是强大的算法设计技巧，它给解提供一个起点。它基本上是首先求解一些较简单问题的分治算法的范例，重要的区别在于这些较简单的问题不是原问题的明确的分割。因为子问题反复被求解，所以重要的是将它们的解记录在一个表中而不是重新计算它们。在某些情况下，解可以被改进（虽然这确实不总是明显的而且常常是困难的），而在另一些情况下，动态规划技巧

**453**

则是所知道的最好的处理方法。

从某种意义上讲，如果你看到一个动态规划问题，那么就看到了所有的问题。有关动态规划的更多示例可以在一些练习和参考文献中找到。

## 10.4 随机化算法

假设你是一位教授，正在布置每周的程序设计作业。你想确保学生在完成自己的程序，或者至少理解他们提交上来的程序。一种解决方案是在每个程序提交的当天进行一次测验。另一方面，这些测验花费课外时间，因此实际上只能对大约半数的程序这么做。你的问题是决定什么时候进行这些测验。

当然，如果事先宣布测验，那么这可以解释为对得不到测验的50%程序的默许。于是，可能采取不宣布的策略对挑选的程序进行测验，不过学生很快就会搞清楚这种策略。另一种可能是对看似重要的程序进行测验，而这又会泄露每个学期的类似的测验风格。学生传播都考些什么样的题，这种方法很可能经过一个学期以后就没有什么价值了。

消除这些弊端的一种方法是使用一个硬币。对每一个程序进行测验（举行测验远不如给他们评分消耗时间），在开始上课时教授将掷硬币来决定是否要举行测验。采用这种方式，在上课前不可能知道是否要进行测验，而测验的模式每个学期之间也不重复。这样，不管前面的测验都是什么规律，学生只能预计测验发生的概率为50%。这种方法的缺点是有可能整个学期都没有测验，不过这不太可能发生，除非硬币有问题。每个学期测验的期望次数是程序数目的一半，并且测验的次数将以高概率不会太偏离这个数目。

这个例子叙述了**随机化算法**（randomized algorithm）的方法。在算法期间，随机数至少有一次用于决策。该算法的运行时间不只依赖于特定的输入，而且依赖于所出现的随机数。

随机化算法的最坏情形运行时间几乎总是和非随机化算法的最坏情形运行时间相同。重要的区别在于，好的随机化算法没有不好的输入，而只有不好的随机数（相对于特定的输入）。这看起来似乎只是哲学上的差别，但是实际上它是相当重要的，正如下面的例子所示。

考虑快速排序的两种变形。方法A用第一个元素作为枢纽元，而方法B使用随机选出的元素作为枢纽元。在这两种情形下，最坏情形运行时间都是 $\Theta(N^2)$ ，因为在每一步都有可能选取最大的元素作为枢纽元。两种最坏情形之间的区别在于，存在特定的输入总能够出现在A中并产生不好的运行时间。当每一次给定已排序的数据时，方法A都将以 $\Theta(N^2)$ 时间运行。如果方法B以相同的输入运行两次，那么它将有二个不同的运行时间，这依赖于出现什么样的随机数。

454

在运行时间的计算中我们始终假设所有的输入都是等可能的。实际上这并不成立，例如，几乎已排序的输入要比统计上期望的出现更经常，而这会产生一些问题，特别是对快速排序和二叉查找树。通过使用随机化算法，特定的输入不再重要，重要的是随机数，我们可以得到一个**期望运行时间**，此时是对所有可能的随机数取平均而不是对所有可能的输入求平均。使用随机枢纽元的快速排序算法是一个 $O(M \log N)$ 期望时间算法。这就是说，对任意的输入，包括已经排序的输入，根据随机数统计学理论，运行时间的期望值为 $O(M \log N)$ 。期望运行时间界多少要强于平均时间界，当然要比对应的最坏情形界弱。另一方面，正如我们在选择问题中所看到的，得到最坏情形时间界的那些解决方案不如它们的平均情形那样在实际中常见。但随机化算法却是如此。

本节将讨论随机化的两个用途。首先，介绍以 $O(\log N)$ 期望时间支持二叉查找树操作的新颖的方案。这意味着不存在不好的输入，只有不好的随机数。从理论的观点看，这并没有那么令人振奋，因为平衡查找树在最坏情形下达到了这个界。然而，随机化的使用产生了对查找、插入特

别是删除的相对简单的算法。

第二个应用是测试大数是否是素数的随机化算法。我们介绍的这种算法运行很快，但偶尔会有错。不过，发生错误的概率可以小到忽略不计。

### 10.4.1 随机数生成器

由于该算法需要随机数，因此必须要有一种方法去生成它。实际上，真正的随机性在计算机上是不可能的，因为这些数依赖于算法，因而不可能是随机的。一般说来，产生伪随机数（pseudorandom number）就足够了，伪随机数是看起来像是随机的数。随机数有许多已知的统计性质；伪随机数满足这些性质的大部分。令人惊奇的是，这说起来容易做起来可就难了。

设只需要抛一枚硬币；这样，必然随机地生成0（正面）或1（反面）。一种做法是考察系统时钟，这个时钟可以把时间记录成整数，而这个整数是从某个起始时刻开始计数的秒数。此时可以使用最低的二进制位。问题在于，如果需要的是随机数序列，那么这种方法就不理想了。1s是一个长的时间段，在程序运行时时钟可能根本没变化。即使时间用微秒（ $\mu\text{s}$ ）为单位记录，如果程序自身正在运行，那么所生成的数的序列也远不是随机的，因为在对生成器的多次调用之间的时间在每次程序调用时可能都是一样的。此时可以看到，真正需要的是随机数的序列（sequence）<sup>1</sup>，这些数应该独立地出现。如果一枚硬币抛出后出现的是正面，那么下一次再抛出时出现正面或反面应该还是等可能的。

455

产生随机数的最简单的方法是线性同余数生成器，它于1951年由Lehmer首先描述。数 $x_1, x_2, \dots$ 的生成满足

$$x_{i+1} = Ax_i \bmod M$$

为了开始这个序列，必须给出 $x_0$ 的某个值。这个值叫作种子（seed）。如果 $x_0 = 0$ ，那么这个序列远不是随机的，但是如果 $A$ 和 $M$ 选择得正确，那么任何其他 $1 \leq x_0 < M$ 都是同等有效的。如果 $M$ 是素数，那么 $x_i$ 就决不会是0。作为一个例子，如果 $M = 11$ ， $A = 7$ ，而 $x_0 = 1$ ，那么所生成的数为

$$7, 5, 2, 3, 10, 4, 6, 9, 8, 1, 7, 5, 2, \dots$$

注意，在 $M - 1 = 10$ 个数以后，序列将重复。因此，这个序列的周期为 $M - 1$ ，它尽可能大（根据鸽巢原理）。如果 $M$ 是素数，那么总存在对 $A$ 的一些选择能够给出全周期（full period） $M - 1$ 。对 $A$ 的有些选择则得不到这样的周期；如果 $A = 5$ 而 $x_0 = 1$ ，那么序列有一个短周期5。

$$5, 3, 4, 9, 1, 5, 3, 4, \dots$$

如果 $M$ 选择得很大，比如31位的素数，那么对于大部分的应用来说周期应该是非常长的。Lehmer建议使用31位的素数 $M = 2^{31} - 1 = 2\,147\,483\,647$ 。对于这个素数， $A = 48\,271$ 是给出全周期生成器的许多值中的一个，它的用途已经被深入研究并被这个领域的专家推荐。后面我们将看到，对于随机数生成器，贸然修改通常意味着失败，因此最好还是继续使用这个公式直到有新的研究成果发布。

这像是一个实现起来很简单的例程。类变量一般用来存放 $x$ 的序列的当前值。当调试一个使用随机数的程序的时候，可能最好是置 $x_0 = 1$ ，这使得总是出现相同的随机序列。当程序工作时，可以使用系统时钟，也可以要求用户输入一个值作为种子。

返回一个位于开区间 $(0, 1)$ 的随机实数（0和1是不能取的值）也是很常见的；这可以通过除以

1. 在本节的其余部分我们将使用随机代替伪随机。



$M$ 得到。由此可知，在任意闭区间 $[\alpha, \beta]$ 内的随机数可以通过规范化来计算。这将产生图10-54中“明显的”类，不过，该类是不正确的。

```

1 static const int A = 48271;
2 static const int M = 2147483647;
3
4 class Random
5 {
6     public:
7         explicit Random( int initialValue = 1 );
8
9         int randomInt( );
10        double random0_1( );
11        int randomInt( int low, int high );
12
13    private:
14        int state;
15 };
16
17 /**
18  * Construct with initialValue for the state.
19  */
20 Random::Random( int initialValue )
21 {
22     if( initialValue < 0 )
23         initialValue += M;
24
25     state = initialValue;
26     if( state == 0 )
27         state = 1;
28 }
29
30 /**
31  * Return a pseudorandom int, and change the
32  * internal state. DOES NOT WORK CORRECTLY.
33  * Correct implementation is in Figure 10.55.
34  */
35 int Random::randomInt( )
36 {
37     return state = ( A * state ) % M;
38 }
39
40 /**
41  * Return a pseudorandom double in the open range 0..1
42  * and change the internal state.
43  */
44 double Random::random0_1( )
45 {
46     return (double) randomInt( ) / M;
47 }

```

图10-54 不能正常工作的随机数生成器

这个类的问题是乘法可能溢出；虽然这不是错误，但是它影响计算的结果，因而影响伪随机性。Schrage给出一个过程，在这个过程中所有的计算均可在32位机上进行而不会溢出。计算 $M/A$ 的商和余数，并把它们分别定义为 $Q$ 和 $R$ 。在上述情况下， $Q = 44\,488$ ， $R = 3399$ ， $R < Q$ 。我们有



456  
457

$$\begin{aligned}
 x_{i+1} &= Ax_i \bmod M = Ax_i - M \left\lfloor \frac{Ax_i}{M} \right\rfloor \\
 &= Ax_i - M \left\lfloor \frac{x_i}{Q} \right\rfloor + M \left\lfloor \frac{x_i}{Q} \right\rfloor - M \left\lfloor \frac{Ax_i}{M} \right\rfloor \\
 &= Ax_i - M \left\lfloor \frac{x_i}{Q} \right\rfloor + M \left( \left\lfloor \frac{x_i}{Q} \right\rfloor - \left\lfloor \frac{Ax_i}{M} \right\rfloor \right)
 \end{aligned}$$

由于  $x_i = Q \left\lfloor \frac{x_i}{Q} \right\rfloor + x_i \bmod Q$ , 因此可以代入到右边的第一项  $Ax_i$  并得到

$$\begin{aligned}
 x_{i+1} &= A \left( Q \left\lfloor \frac{x_i}{Q} \right\rfloor + x_i \bmod Q \right) - M \left\lfloor \frac{x_i}{Q} \right\rfloor + M \left( \left\lfloor \frac{x_i}{Q} \right\rfloor - \left\lfloor \frac{Ax_i}{M} \right\rfloor \right) \\
 &= (AQ - M) \left\lfloor \frac{x_i}{Q} \right\rfloor + A(x_i \bmod Q) + M \left( \left\lfloor \frac{x_i}{Q} \right\rfloor - \left\lfloor \frac{Ax_i}{M} \right\rfloor \right)
 \end{aligned}$$

由于  $M = AQ + R$ , 因此  $AQ - M = -R$ . 于是得到

$$x_{i+1} = A(x_i \bmod Q) - R \left\lfloor \frac{x_i}{Q} \right\rfloor + M \left( \left\lfloor \frac{x_i}{Q} \right\rfloor - \left\lfloor \frac{Ax_i}{M} \right\rfloor \right)$$

项  $\delta(x_i) = \left\lfloor \frac{x_i}{Q} \right\rfloor - \left\lfloor \frac{Ax_i}{M} \right\rfloor$  或者是0, 或者是1, 因为两项都是整数而它们的差在0和1之间。因此, 我们有

$$x_{i+1} = A(x_i \bmod Q) - R \left\lfloor \frac{x_i}{Q} \right\rfloor + M\delta(x_i)$$

快速验证表明, 因为  $R < Q$ , 故所有的余项均可计算而没有溢出 (这就是选择  $A = 48\,271$  的原因之一)。此外, 仅当余项的值小于0时  $\delta(x_i) = 1$ , 因此  $\delta(x_i)$  不需要显式地计算而是可以通过简单的测试来确定。于是得到图10-55中修正后的程序。

```

1 static const int A = 48271;
2 static const int M = 2147483647;
3 static const int Q = M / A;
4 static const int R = M % A;
5
6 /**
7  * Return a pseudorandom int, and change the internal state.
8  */
9 int Random::randomInt( )
10 {
11     int tmpState = A * ( state % Q ) - R * ( state / Q );
12
13     if( tmpState >= 0 )
14         state = tmpState;
15     else
16         state = tmpState + M;
17
18     return state;
19 }

```

图10-55 不溢出的随机数生成器

人们可能会想到要假设所有的机器在标准库中都有一个至少像图10-55中的程序那么好的随机数生成器，但是，情况并非如此。许多库中的生成器基于函数 458

$$x_{i+1} = (Ax_i + C) \bmod 2^B$$

其中 $B$ 的选取要匹配机器整数的位数，而 $C$ 是奇数。但是，这些生成器总是产生在奇偶之间交替的 $x_i$ 的值——很难具有理想的性质。事实上，低 $k$ 位（充其量）是以周期 $2^k$ 循环。许多其他随机数生成器要比图10-55所提供的随机数生成器的循环小得多，这些生成器对于需要长的随机数序列的情况是不合适的。UNIX的drand48函数使用这种形式的生成器。不过，它使用48位线性同余生成器并且只返回高32位，这样，避免了在低阶位上的循环问题。用到的常数是 $A=25\,214\,903\,917$ 、 $B=48$ 以及 $C=13$ 。最后，通过添加一个常数到方程中去可能会得到更好的随机数生成器。例如，

$$x_{i+1} = (48\,271x_i + 1) \bmod (2^{31} - 1)$$

多少会更加随机一些。这说明这些随机数生成器是多么的脆弱。

$$[48\,271(179\,424\,105) + 1] \bmod (2^{31} - 1) = 179\,424\,105$$

因此，如果种子是179 424 105，那么生成器将陷入周期为1的循环。

### 10.4.2 跳跃表

随机化的第一个用途是以 $O(\log N)$ 期望时间支持查找和插入的数据结构。正如在本节介绍中所提到的，这意味着对于任意输入序列的每一次操作，其运行时间都有期望值 $O(\log N)$ ，其中的期望是基于随机数生成器的。能够完成添加删除以及所有涉及排序的操作，并且能够得到与二叉查找树的平均时间界匹配的期望时间界。

最简单的支持查找的可能的数据结构是链表。图10-56是一个简单的链表。执行一次查找的时间正比于必须考察的结点数，结点数最多是 $N$ 。

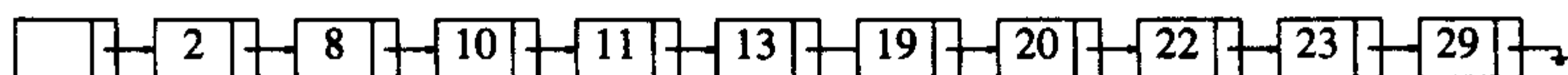


图10-56 简单链表

图10-57表示一个链表，在该链表中，每隔一个结点都有一个附加的到它在表中前两个位置上的结点的链。正因为如此，在最坏情形下，最多考察 $\lceil N/2 \rceil + 1$ 个结点。

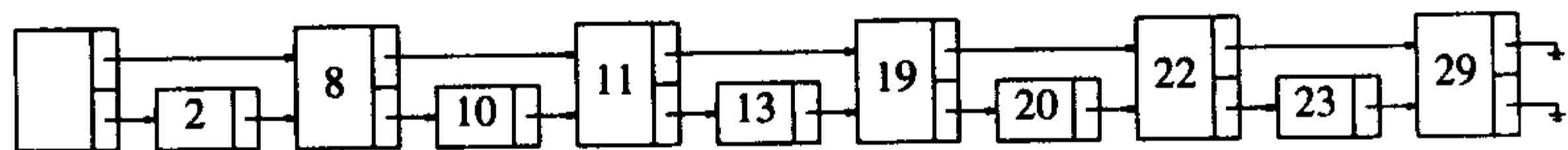


图10-57 带有到前面两个表元素的链的链表

扩展这种想法，得到图10-58。这里，每隔三个结点都有一个到该结点前四个位置上的结点的链。只有 $\lceil N/4 \rceil + 2$ 个结点被考察。

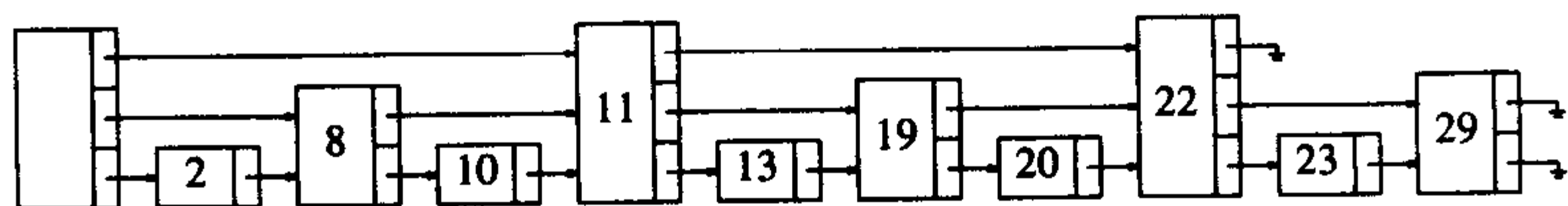
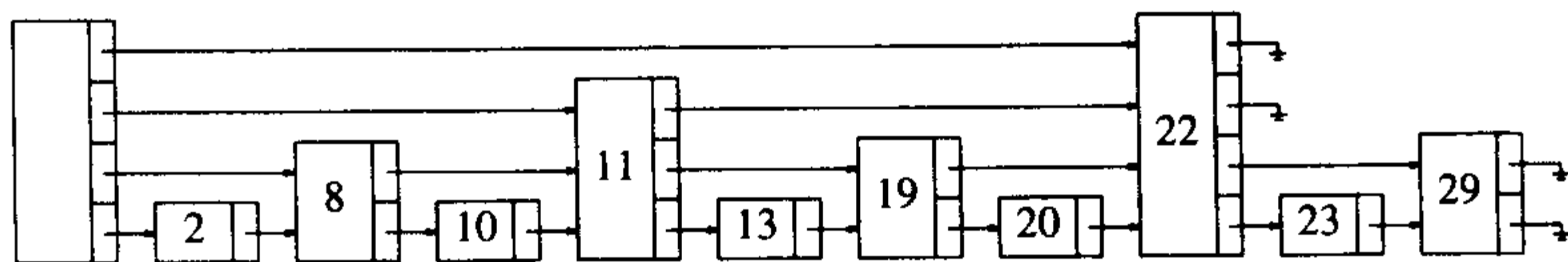


图10-58 带有到前面四个表元素的链的链表

这种跳跃幅度的一般情形如图10-59所示。每个第 $2^i$ 结点都有一个到这个结点前面的 $2^i$ 个结

459

点。链的总数仅是原始链表的两倍，但现在在一次查找中最多考察 $\lceil \log N \rceil$ 个结点。不难看出，一次查找总的时间消耗为 $O(\log N)$ ，这是因为查找由向前到一个新的结点或者在同一结点下降到低一级的链组成。在一次查找期间每一步总的时间消耗最多为 $O(\log N)$ 。注意，在这种数据结构中的查找基本上是二分搜索（binary search）。

图10-59 带有到前面 $2^i$ 个表元素的链的链表

这种数据结构的问题是有效的插入太过于呆板。使这种数据结构可用的关键是稍微放松结构条件。我们将带有 $k$ 个链的结点定义为 $k$ 阶结点（level  $k$  node）。如图10-59所示，任意 $k$ 阶结点上的第 $i$ 阶（ $k \geq i$ ）链链接的下一个结点至少具有 $i$ 阶。这是一个容易保留的性质；不过，图10-59指出比它限制性更强的性质。这样，把链接到前面 $2^i$ 个结点的第 $i$ 个链这种限制去掉，而代之以上面稍松一些的限制条件。

需要插入新元素的时候，为它分配一个新的结点。此时，必须决定该结点是多少阶的。考察图10-59可以发现，大约一半的结点是1阶结点，大约1/4的结点是2阶结点，一般地，大约 $1/2^i$ 的结点是 $i$ 阶结点。按照这个概率分布随机选择结点的阶数。最容易的方法是抛一枚硬币直到正面出现，把抛硬币的总次数作为该结点的阶数。图10-60显示了一个典型的跳跃表。

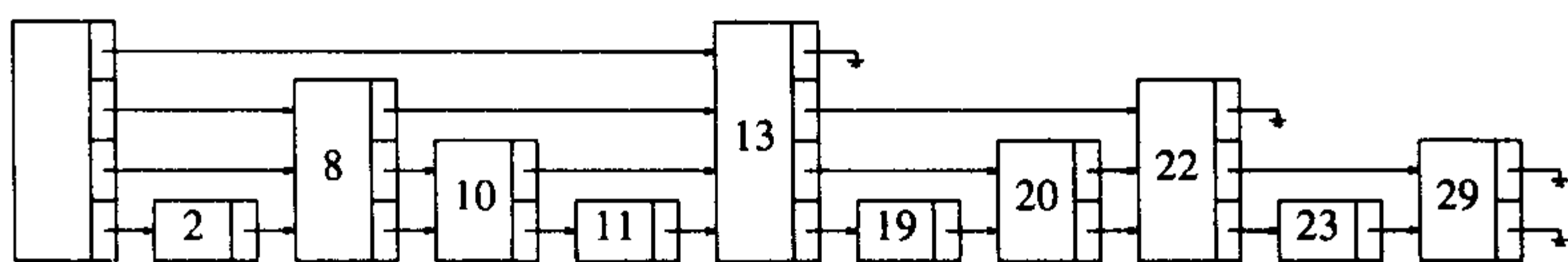


图10-60 一个跳跃表

460

给出上面的分析以后，跳跃表算法的描述就简单了。为执行一次查找，在头结点从最高阶的链开始，沿着这个阶一直走，直至发现大于正在寻找的结点的下一个结点（或者是null）。这时，转到下一个低一阶的阶并继续这种策略。当进行到一阶停止时，或者位于正在寻找的结点的前面，或者它不在这个表中。为了执行一次insert，像查找时那样进行，始终记住每一个使我们转到下一阶的结点。最后，将新结点（它的阶是随机确定的）拼接入表中。操作见图10-61。

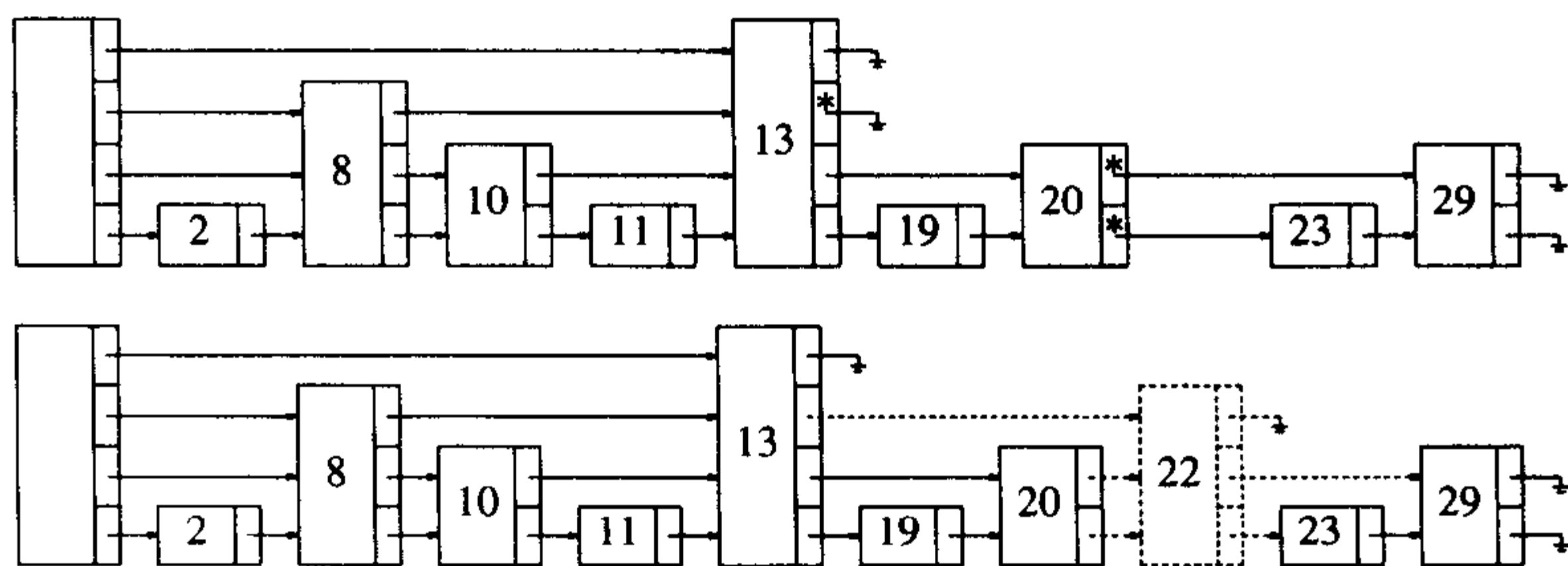


图10-61 插入前和插入后的跳跃表

粗略分析表明，由于在原（非随机化的）算法上没有改变每一阶的结点的期望个数，因此预计穿越同阶上的结点的总的工作量是不变的。这告诉我们，这些操作具有期望（价）开销值

$O(\log N)$ 。当然，更正式的证明是需要的，但它与此没有太大的区别。

跳跃表类似于散列表，它们都需要估计表中的元素个数（从而确定阶的数目）。如果得不到这种估计，那么可以假设一个大的数或者使用一种类似于再散列的方法。经验表明，跳跃表同许多平衡查找树实现方法一样有效，而且确实用许多语言实现都会简单得多。

### 10.4.3 素性测试

本节将考察确定一个大数是否是素数的问题。正如第2章末尾谈到的，某些密码方案依赖于大数分解的难度，比如将一个400位数分解成两个100位的素数。为了实现这种方案，需要一种生成两个大素数的方法。如果 $d$ 是 $N$ 的数字位数测试被从3到 $\sqrt{N}$ 的奇数整除的明显的方法大约需要  $\frac{1}{2}\sqrt{N}$  次除法，它大约为 $10^{d/2}$ ，并且对200位的数完全不实用。

461

本节将给出一个可以测试素性的多项式时间算法。如果这个算法宣称一个数不是素数，那么可以肯定这个数不是素数。如果该算法宣称一个数是素数，那么，这个数将很可能（以高的概率）但不是100%是素数。错误的概率不依赖于被测试的特定的数，而是依赖于由算法做出的随机选择。因此，这个算法偶尔会出错，不过我们将会看到，可以让出错的几率任意小。

算法的关键是著名的费马（Fermat）定理。

**定理10.10（费马小定理）** 如果 $P$ 是素数，且 $0 < A < P$ ，那么 $A^{P-1} \equiv 1 \pmod{P}$ 。

**证明** 这个定理的证明可以在任一本有关数论的教科书中找到。 ■

例如，由于67是素数，因此 $2^{66} \equiv 1 \pmod{67}$ 。这提出了测试一个数 $N$ 是否是素数的算法：只要检验一下是否 $2^{N-1} \equiv 1 \pmod{N}$ 。如果 $2^{N-1} \not\equiv 1 \pmod{N}$ 不成立，那么可以肯定 $N$ 不是素数。另一方面，如果等式成立，那么 $N$ 很可能是素数。例如，满足 $2^{N-1} \equiv 1 \pmod{N}$ 但不是素数的最小的 $N$ 是341。

这个算法偶尔会出错，但问题是它总出一些相同的错误。换句话说，存在 $N$ 的一个固定的集合，对于这个集合该方法行不通。可以尝试将该算法按下面的方式随机化：随机取 $1 < A < N-1$ 。如果 $A^{N-1} \equiv 1 \pmod{N}$ ，声明 $N$ 可能是素数，否则声明 $N$ 肯定不是素数。如果 $N=341$ 而 $A=3$ ，那么 $3^{340} \equiv 56 \pmod{341}$ 。因此，如果算法碰巧选择 $A=3$ ，那么对于 $N=341$ 将得到正确的答案。

虽然这看起来没有问题，但是却存在一些数，对于 $A$ 的大部分选择它们甚至可以骗过该算法。一种这样的数集称为Carmichael数（Carmichael number），这些数不是素数但对所有与 $N$ 互素的 $0 < A < N$ 却满足 $A^{N-1} \equiv 1 \pmod{N}$ 。最小的这样的数是561。因此，我们还需要一个附加的测试来改进不出错的几率。

在第7章，我们证明过一个关于平方探测（quadratic probing）的定理。这个定理的特殊情况如下。

**定理10.11** 如果 $P$ 是素数且 $0 < X < P$ ，那么 $X^2 \equiv 1 \pmod{P}$ 仅有的解为 $X=1$ ， $P-1$ 。

**证明**  $X^2 \equiv 1 \pmod{P}$ 意味着 $X^2 - 1 \equiv 0 \pmod{P}$ 。这就是说， $(X-1)(X+1) \equiv 0 \pmod{P}$ 。由于 $P$ 是素数， $0 < X < P$ ，因此 $P$ 必然整除 $(X-1)$ 或者 $(X+1)$ ，由此推出定理。 ■

因此，如果在计算 $A^{N-1} \pmod{N}$ 的任一时刻我们发现违背了该定理，那么可以断言 $A$ 肯定不是素数。如果使用2.4.4节的方法pow，那么将有几种机会来应用这种测试。可以修改对 $N$ 求余运算的例程并应用定理10.11的测试，这种策略在图10-62中以伪代码的形式实现。

462



```

1  /**
2  * Function that implements the basic primality test.
3  * If witness does not return 1, n is definitely composite.
4  * Do this by computing a^i (mod n) and looking for
5  * non-trivial square roots of 1 along the way.
6  */
7  HugeInt witness( const HugeInt & a, const HugeInt & i, const HugeInt & n )
8  {
9      if( i == 0 )
10         return 1;
11
12     HugeInt x = witness( a, i / 2, n );
13     if( x == 0 )    // If n is recursively composite, stop
14         return 0;
15
16     // n is not prime if we find a non-trivial square root of 1
17     HugeInt y = ( x * x ) % n;
18     if( y == 1 && x != 1 && x != n - 1 )
19         return 0;
20
21     if( i % 2 != 0 )
22         y = ( a * y ) % n;
23
24     return y;
25 }
26
27 /**
28 * The number of witnesses queried in randomized primality test.
29 */
30 const int TRIALS = 5;
31
32 /**
33 * Randomized primality test.
34 * Adjust TRIALS to increase confidence level.
35 * n is the number to test.
36 * If return value is false, n is definitely not prime.
37 * If return value is true, n is probably prime.
38 */
39 bool isPrime( const HugeInt & n )
40 {
41     Random r;
42
43     for( int counter = 0; counter < TRIALS; counter++ )
44         if( witness( r.randomInt( 2, (int) n - 2 ), n - 1, n ) != 1 )
45             return false;
46
47     return true;
48 }

```

463

图10-62 一种概率素性测试算法（伪代码）

我们知道，如果witness返回任何不是1的数，那么它就已经证明了 $N$ 不可能是素数，其证明是非构造性的，因为它并没有具体给出找到因子的方法。业已证明，对于任何（充分大的） $N$ ，至多有 $A$ 的 $(N-9)/4$ 个值会使本算法得出错误的结论。因此，如果 $A$ 是随机选取的，而且算法的结论是 $N$ （很可能）为素数，那么至少有75%的概率算法是正确的。设witness运行50次，而算法得出错误结论的概率最多是 $\frac{1}{4}$ 。因此，50次独立的随机试验使算法出错的概率决不会超过 $1/4^{50} = 2^{-100}$ 。实际上这是非常保守的估计，它只对 $N$ 的某些选择成立。即使如此，人们更可能

看到的是硬件的错误，而不是数的素性不正确的宣布结果。

素性测试的随机化算法非常重要，因为它们明显比最佳非随机化算法快。而且，虽然随机化算法偶尔会产生错误的判断，但这种机会非常小，足可忽略。

很多年来，可能以 $d$ 的多项式时间确定性地测试一个 $d$ 位数的素性一直是令人怀疑的，但是并不知道这种算法。不过，最近素性测试的确定性的多项式时间算法已经发现。然而，这些算法还只是令人激动的理论结果，它们还不能与随机化方法竞争。本章末尾的参考文献提供了更多的信息。

## 10.5 回溯算法

我们将要考察的最后一个算法设计技巧是回溯（backtracking）。在许多情况下，回溯算法相当于穷举搜索的巧妙实现，但性能一般不理想。不过，情况并非总是如此，即使是如此，在某些情形下它相对于蛮力穷举搜索，工作量也有显著的节省。当然，性能是相对的：对于排序而言， $O(N^2)$ 的算法是相当差的，但对旅行商（或任何NP完全）问题， $O(N^5)$ 算法则是里程碑式的结果。

回溯算法的一个具体例子是在一套新房子内摆放家具的问题。存在许多尝试的可能性，但一般只有一些是具体要考虑的。开始什么也不摆放，然后是每件家具被摆放在室内的某个位置。如果所有的家具都已摆好而且户主很满意，那么算法终止。如果摆到某一步，该步之后的所有家具摆放方法都不理想，那么必须撤销这一步并尝试另外的摆放方法。当然，这也可能导致另外的撤销，等等。如果我们发现撤销了所有可能的第一步摆放位置，那么就不存在满意的家具摆放方法。否则，最终将终止在满意的摆放位置上。注意，虽然这个算法基本上是蛮力的，但是它并不直接尝试所有的可能。例如，把沙发放进厨房的各种摆法是决不会尝试的。许多其他不好的摆放方法早就取消了，因为令人讨厌的摆放的子集是已知的。在一步内删除一大组可能性的做法叫作裁剪（pruning）。

下面将给出回溯算法的两个例子。第一个例子是计算几何中的问题，第二个例子阐述在诸如国际象棋和西洋跳棋的对弈中计算机如何选取行棋的步骤。

464

### 10.5.1 公路收费点重建问题

设给定 $N$ 个点 $p_1, p_2, \dots, p_N$ ，它们位于 $x$ 轴上。 $x_i$ 是 $p_i$ 点的 $x$ 坐标。进一步假设 $x_1 = 0$ ，并且这些点从左到右给出。这 $N$ 个点确定在每一对点间的 $N(N-1)/2$ 个（不必是唯一的）形如 $|x_i - x_j|$ （ $i \neq j$ ）的距离 $d_1, d_2, \dots, d_N$ 。显然，如果给定点集，那么容易以 $O(N^2)$ 时间构造距离的集合。这个集合不是排好序的，但是如果愿意花 $O(N^2 \log N)$ 时间界整理，那么这些距离也可以被排序。公路收费点重建问题（turnpike reconstruction problem）是从这些距离重建一个点集，它在物理学和分子生物学（参见有关该信息更专门的参考文献）中都有应用。这个名称源于对美国东海岸高速公路上那些收费公路出口的模拟。正像大数分解比乘法困难一样，重建问题也比建造问题困难。没有人能够给出一个算法，保证在多项式时间内完成计算。我们将要介绍的算法一般以 $O(N^2 \log N)$ 运行，但在最坏情形下可能要花费指数时间。

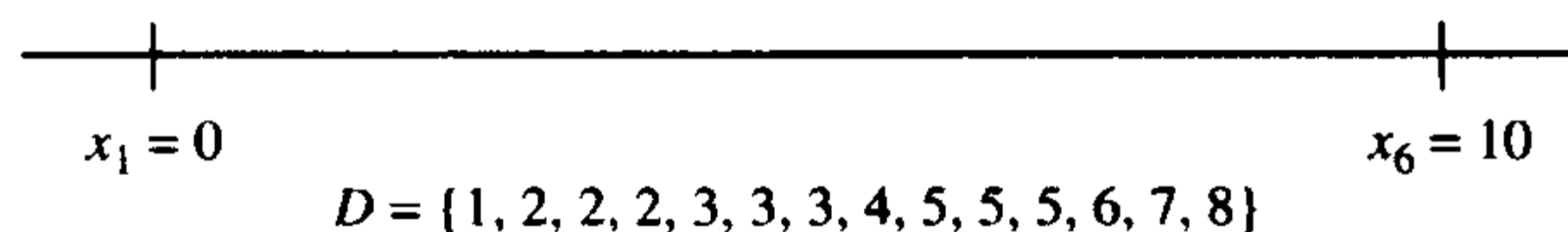
当然，若给定该问题的一个解，则可以通过对所有的点加上一个偏移量来构建无穷多其他的解。这就是我们坚持要将第一个点置于0处以及构建解的点集以非递减顺序输出的原因。

令 $D$ 是距离的集合，并设 $|D| = M = N(N-1)/2$ 。作为例子，设

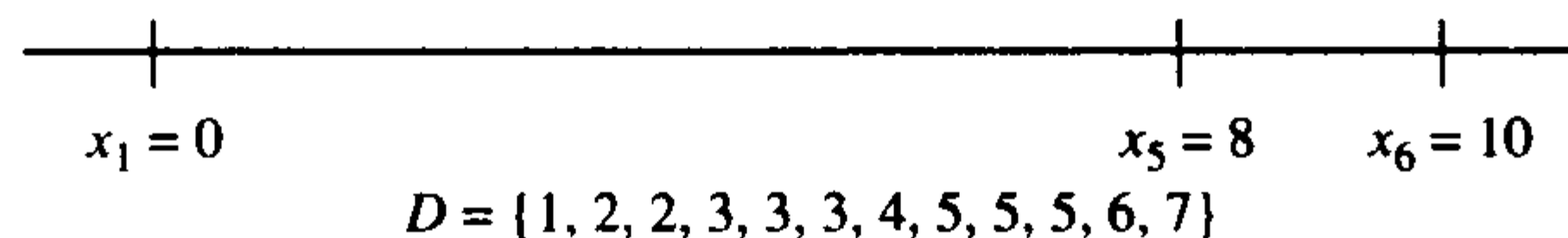
$$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$$

由于 $|D| = 15$ ，因此 $N = 6$ 。算法以置 $x_1 = 0$ 开始。显然， $x_6 = 10$ ，因为10是 $D$ 中最大的元素。将10从

$D$ 中删除, 我们放置的点和剩下的距离如下图所示。



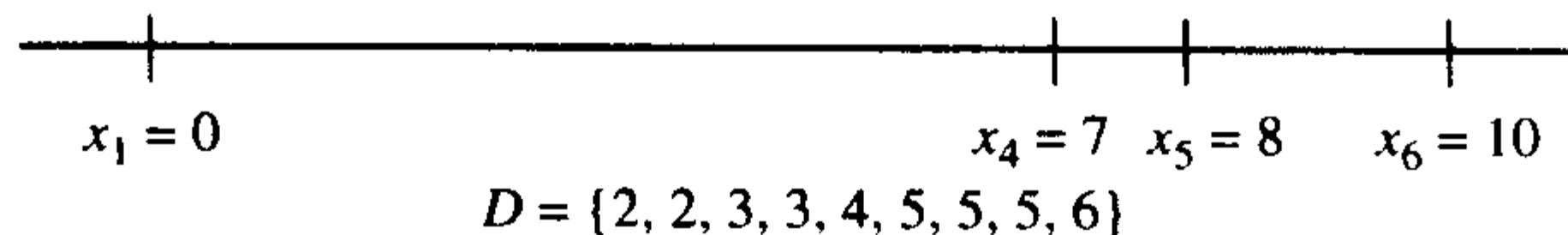
剩下的距离中最大的是8, 这就是说, 或者 $x_2 = 2$ , 或者 $x_5 = 8$ 。由对称性可以断定这种选择是不重要的, 因为或者两个选择都产生解(它们互为镜像), 或者都不产生最终的解, 所以可置 $x_5 = 8$ 而不至于影响问题的解。然后从 $D$ 中删除距离 $x_6 - x_5 = 2$ 和 $x_5 - x_1 = 8$ , 得到



下一步是不明显的。由于7是 $D$ 中最大的数, 因此或者 $x_4 = 7$ , 或者 $x_2 = 3$ 。如果 $x_4 = 7$ , 那么距离 $x_6 - 7 = 3$ 和 $x_5 - 7 = 1$ 也必须出现在 $D$ 中; 我们一看便知它们确实在 $D$ 中。另一方面, 如果置 $x_2 = 3$ , 那么 $3 - x_1 = 3$ 和 $x_5 - 3 = 5$ 就必须在 $D$ 中; 这些距离也的确在 $D$ 中。因此, 我们不对哪种选择做强求。这样, 尝试其中的一种看它是否产生问题的解。如果不行, 那么退回来再尝试另外的选择。

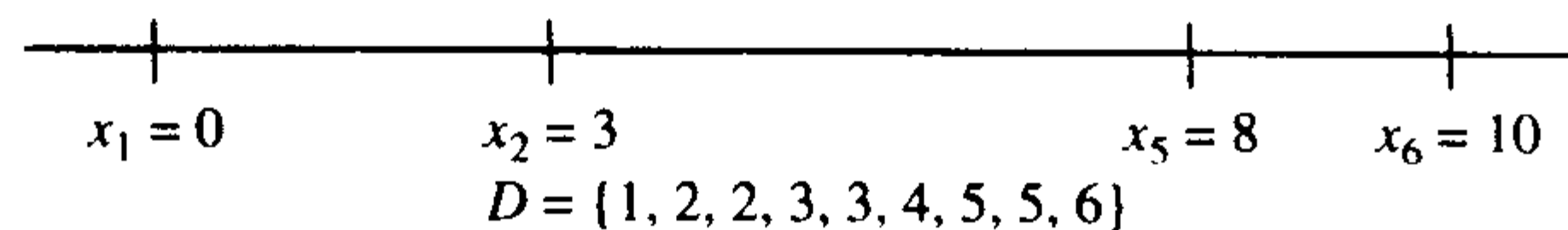
465

尝试第一个选择, 置 $x_4 = 7$ , 得到

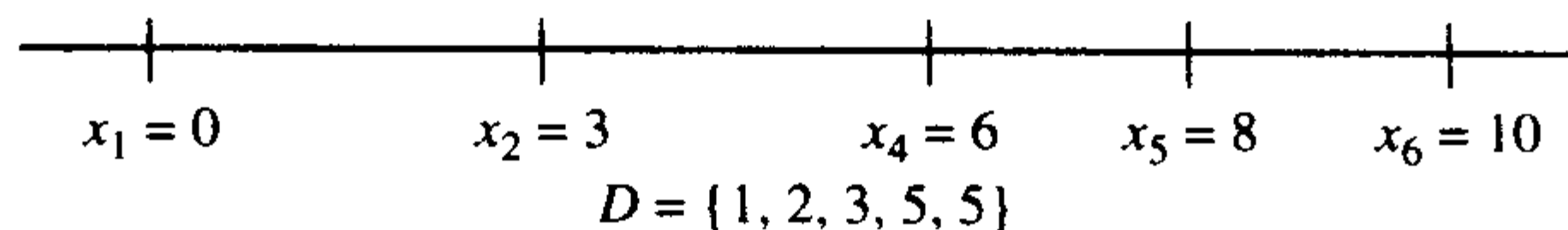


此时, 我们得到 $x_1 = 0, x_4 = 7, x_5 = 8$ 和 $x_6 = 10$ 。现在最大的距离是6, 因此或者 $x_3 = 6$ , 或者 $x_2 = 4$ 。但是, 如果 $x_3 = 6$ , 那么 $x_4 - x_3 = 1$ , 这是不可能的, 因为1不再属于 $D$ 。另一方面, 如果 $x_2 = 4$ , 那么 $x_2 - x_1 = 4$ 和 $x_5 - x_2 = 4$ , 这也是不可能的, 因为4只在 $D$ 中出现一次。因此, 这个推导思路得不到解, 我们需要回溯。

由于 $x_4 = 7$ 不能产生解, 因此尝试 $x_2 = 3$ 。如果这也不行, 那么停止计算并报告无解。现在, 我们有



我们必须再一次在 $x_4 = 6$ 和 $x_3 = 4$ 之间选择。 $x_3 = 4$ 是不可能的, 因为 $D$ 中只出现一个4, 而该选择意味着要有两个。 $x_4 = 6$ 是可能的, 于是得到



唯一剩下的选择是 $x_3 = 5$ , 这是可以的, 因为它使得 $D$ 成为空集, 因此我们得到问题的一个解。

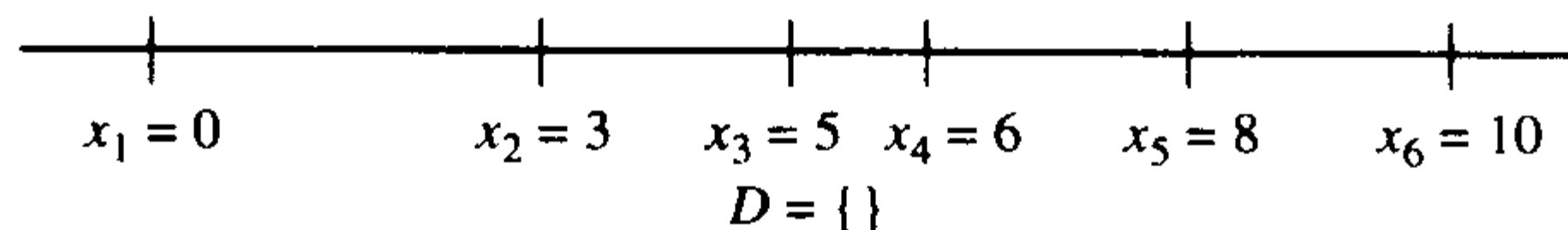


图10-63是一棵决策树, 代表为得到解而采取的行动。这里, 没有对分支进行标记, 而是把

标记放在了分支的目的结点上。带有一个星号的结点表示所选的点与给定的距离不一致；带有两个星号的结点只有不可能的结点作为儿子，因此表示一条不正确的路径。

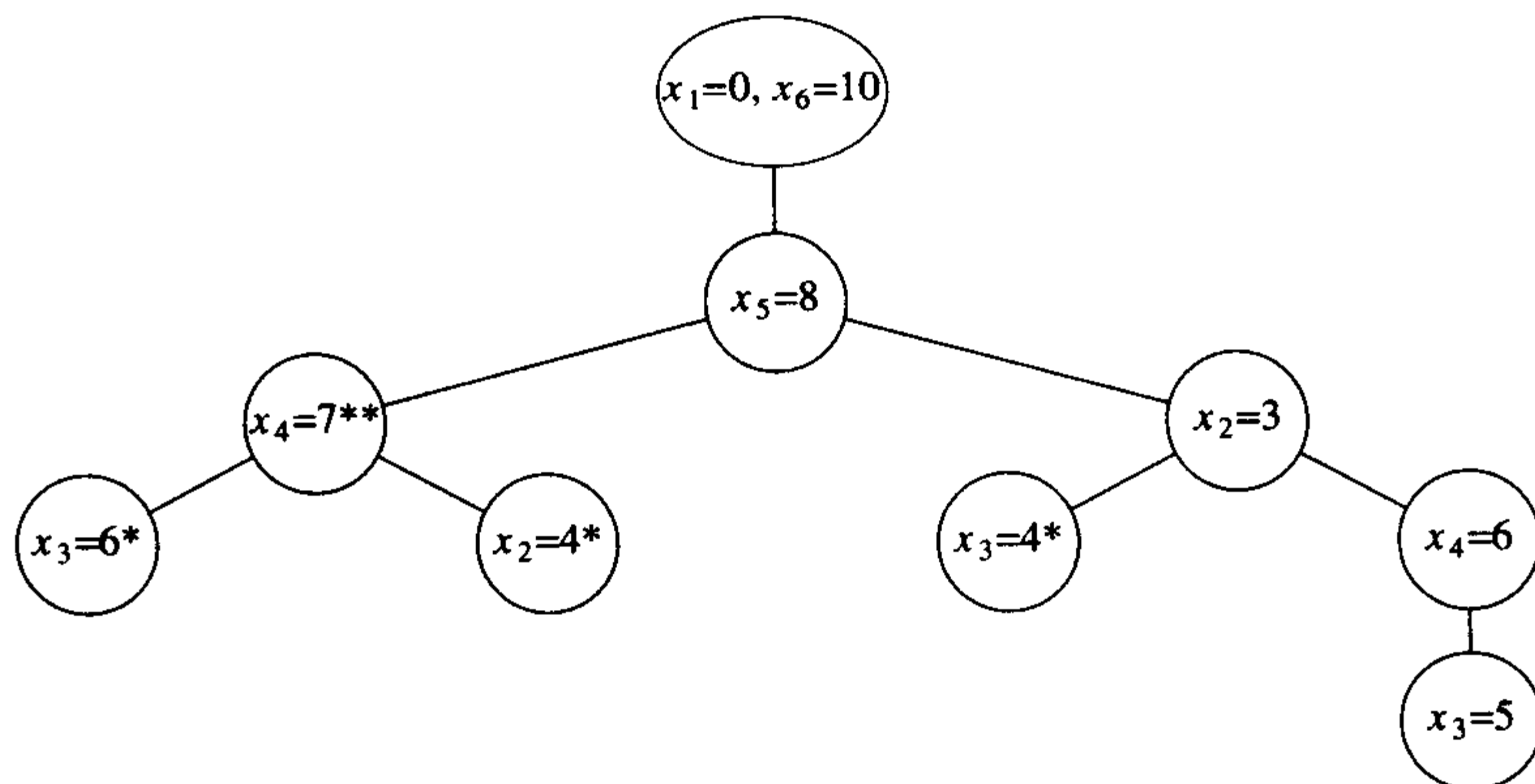


图10-63 公路收费点重建问题的决策树

实现这个算法的伪代码大部分都很简单。驱动例程turnpike如图10-64所示。它接收点的数组 $x$ （不需要初始化）、距离的集合 $D$ 和 $N$ <sup>1</sup>。如果找到一个解，则返回true，答案将被放到 $x$ 中，而 $D$ 将是空集。否则，返回false， $x$ 将是未定义的，距离集合 $D$ 将是未触及的。该例程如上所述设置了 $x_1$ 、 $x_{N-1}$ 和 $x_N$ ，修改了 $D$ ，并且调用了回溯算法place以放置其余的点。假设为保证 $|D|=N(N-1)/2$ 已经进行了检验。

466

```

bool turnpike( vector<int> & x, DistSet d, int n )
{
    x[ 1 ] = 0;
    d.deleteMax( x[ n ] );
    d.deleteMax( x[ n - 1 ] );
    if( x[ n ] - x[ n - 1 ] ∈ d )
    {
        d.remove( x[ n ] - x[ n - 1 ] );
        return place( x, d, n, 2, n - 2 );
    }
    else
        return false;
}

```

图10-64 公路收费点重建算法：驱动例程（伪代码）

更困难的部分是回溯算法，如图10-65所示。与大多数回溯算法一样，最方便的实现方法是递归。我们传递同样的参数以及界 $Left$ 和 $Right$ ； $x_{Left}, \dots, x_{Right}$ 是试图放置的点的 $x$ 坐标。如果 $D$ 是空集（或 $Left > Right$ ），那么解已经找到，可以返回。否则，首先尝试使 $x_{Right} = d_{\max}$ 。如果所有适当的距离都（以正确的值）出现，那么尝试性地放上这一点，删除相应的距离，并尝试从 $Left$ 到 $Right-1$ 填入。如果这些距离不出现，或者尝试从 $Left$ 到 $Right-1$ 填入失败，那么尝试置 $x_{Left} = x_N - d_{\max}$ ，使用类似的方法。如果这样不行，则问题无解；否则，已经找到一个解，而这个信息最终通过return语句和 $x$ 数组传递回turnpike。

467

1. 为使所举的例子方便起见，我们使用了单字母变量名，一般说来这不是好习惯。为了简单，也没给出变量的类型。最后，我们让数组下标从1开始，而不是从0。



```

/**
 * Backtracking algorithm to place the points x[left] ... x[right].
 * x[1]...x[left-1] and x[right+1]...x[n] already tentatively placed.
 * If place returns true, then x[left]...x[right] will have values.
 */
bool place( vector<int> & x, DistSet d, int n, int left, int right )
{
    int dmax;
    bool found = false;

1   if( d.isEmpty( ) )
2       return true;

3   dmax = d.findMax( );

    // Check if setting x[right] = dmax is feasible.
4   if( | x[j] - dmax | ∈ d for all 1≤j<left and right<j≤n )
    {
5       x[right] = dmax;           // Try x[right]=dmax
6       for( 1≤j<left, right<j≤n )
7           d.remove( | x[j] - dmax | );
8       found = place( x, d, n, left, right-1 );

9       if( !found )           // Backtrack
10          for( 1≤j<left, right<j≤n ) // Undo the deletion
11              d.insert( | x[j] - dmax | );
    }

    // If first attempt failed, try to see if setting
    // x[left]=x[n]-dmax is feasible.
12   if( !found && ( | x[n] - dmax - x[j] | ∈ d
13               for all 1≤j<left and right<j≤n ) )
    {
14       x[left] = x[n] - dmax;    // Same logic as before
15       for( 1≤j<left, right<j≤n )
16           d.remove( | x[n] - dmax - x[j] | );
17       found = place( x, d, n, left+1, right );

18       if( !found )           // Backtrack
19          for( 1≤j<left, right<j≤n ) // Undo the deletion
20              d.insert( | x[n] - dmax - x[j] | );
    }

21   return found;
}

```

图10-65 公路收费点重建算法：回溯的步骤（伪代码）

算法的分析涉及两个因素。设第9行到第11行以及第18行到第20行从未执行。可以把 $D$ 作为平衡二叉查找（或伸展）树保存（当然，这需要对代码做些修改）。如果我们从未回溯，那么最多有 $O(N^2)$ 次操作涉及 $D$ ，如在第4行、第12行到第13行中蕴含的删除和一些contains。显然这是对删除提出的，因为 $D$ 有 $O(N^2)$ 个元素而且没有元素被重新插入。每次对place的调用最多用到 $2N$ 次contains，而且由于place在该分析中从未回溯，因此最多有 $2N^2$ 次contains。于是，如果没有回溯，那么运行时间为 $O(N^2 \log N)$ 。

当然，回溯是要发生的。如果回溯反复发生，那么算法的性能就会受到影响。可以通过构建病态的情形迫使它发生。经验表明，如果点的整数坐标在 $[0, D_{\max}]$ 均匀地和随机地分布，其中 $D_{\max} = \Theta(N^2)$ ，那么几乎可以肯定，在整个算法期间最多执行一次回溯。

## 10.5.2 博弈

作为最后一个应用，我们将考虑计算机可能用来进行战略游戏的策略，如西洋跳棋或国际象棋。作为一个例子，我们将使用简单得多的三连游戏棋（tic-tac-toe），因为它使得想法更容易表述。

如果双方都玩到最优，那么三连游戏棋就是平局。通过对逐个情况进行仔细的分析，构造一个从不输棋而且当机会出现时总能赢棋的算法并不是困难的事。这之所以能够做到是因为一些位置是已知的陷阱，可以通过查询表来处理。另外一些方法，如当中央的方格可用时占据该方格，可以使得分析更简单。如果完成了分析，那么通过使用一个表我们总可以只根据当前位置选择一步棋。当然，这种方法需要程序员而不是计算机来进行大部分的思考。

### 1. 极小极大策略

更一般的策略是使用一个求值函数来对一个位置的“好坏”量化。能使计算机获胜的位置可以得到值+1；平局可得到0；使计算机输棋的位置得到值-1。通过考察盘面能够确定这局棋输赢的位置叫作**终端位置**（terminal position）。

如果一个位置不是终端位置，那么该位置的值通过递归地假设双方最优棋步而确定。这叫作**极小极大策略**，因为下棋的一方（人）试图使这个位置的值极小化，而另一方（计算机）却要使它的值极大化。

位置 $P$ 的**后继位置**（successor position）是通过从 $P$ 走一步棋可以达到的任何位置 $P_s$ 。如果在某个位置 $P$ 计算机要走棋，那么它递归地求出所有后继位置的值。计算机选择具有最大值的一步棋；这就是 $P$ 的值。为了得到任意后继位置 $P_s$ 的值，要递归地算出 $P_s$ 的所有后继位置的值，然后选取其中最小的值。这个最小值代表下棋人一方最赞成的应招。

图10-66中的程序使得计算机的策略更清楚。第14行到第18行直接给赢棋或平局赋值。如果这两种情况都不适用，那么这个位置就是非终端位置。注意到value应该包括所有可能后继位置的最大值，第21行把它初始化为最小的可能值，第22行到第37行的循环则为了改进而进行搜索。每一个后继位置递归地依次由第26到第28行算出值来。因为我们将看到过程findHumanMove调用findCompMove，所以这是递归的。如果下棋人对一步棋的应招给计算机留下比计算机在前面最佳棋步所得到的位置更好的位置，那么value和bestMove将被更新。图10-67显示的是下棋人选择棋步的方法。除了下棋人选择的棋步导致最低值的位置外，所有的逻辑实际上都是相同的。事实上，通过传递一个附加的变量不难把这两个过程合并成一个，这个附加变量指出该轮到谁走棋。这样一来确实使得程序多少有些难于读懂，因此我们保留两个分开的例程。

我们把一些支持例程留作练习。代价最高的计算是需要计算机开局的情形。由于在这个阶段棋局处于平局的形势，因此计算机选择方格1。<sup>1</sup>需要考察的位置总共有97 162个，计算要花费几秒。这里没有优化程序的打算。如果下棋人选择中央方格，那么当计算机走第二步棋的时候，所要考察的位置是5185个，当下棋人选择角上的方格时计算机所要考察的位置是9761个，而当下棋人选择非角的边上的方格时计算机要考察13 233个位置。

对于更复杂的游戏，如西洋跳棋和国际象棋，搜索到终端结点的全部棋步显然是不可行的。<sup>2</sup>在这种情况下，我们在达到递归的某个深度之后只能停止搜索。递归停止处的结点则成为终端结点。这些终端结点的值由一个估计位置的值的函数计算得出。例如，在一个国际象棋程序中，求值函数计量诸如棋子、位置因素的相对量和强度这样一些变量。求值函数对于成功是至关重要的，

468  
469

1. 我们将方格从棋盘左上角开始向右编号。不过，这只对支持例程是重要的。

2. 据估计，假如对下棋进行这种搜索，那么对于第一步棋至少需要  $10^{100}$  个位置需要考查。即使是本节稍后描述的改进方法结合使用，这个数字也不能降低到使用的水平。

因为计算机的行棋选步是基于将这个函数极大化的。最好的计算机国际象棋程序具有相当复杂的求值函数。

然而，对于计算机下国际象棋，一个最重要的因素看来是程序能够向前看的棋步的数目。有时我们称之为层 (ply)，它等于递归的深度。为了实现这个功能，需要给搜索例程一个附加的参数。

在对弈程序中增加向前看因子的基本方法是提出一些方法，这些方法对更少的结点求值却不丢失任何信息。我们已经看到的一种方法是使用一个表来记录所有已经被计算过值的位置。例如，在搜索第一步棋的过程中，程序将考察图10-68中的一些位置。如果这些位置的值被存储了，那么一个位置在第二次出现时就不必再重新计算；它实质上变成了一个终端位置。记录这些信息的数据结构叫作**置换表** (transposition table)；它几乎总可通过散列来实现。在许多情况下，这可以节省大量的计算。例如，在一盘棋的残局阶段，此时相对来说只有很少的棋子，时间的节省使得一次搜索可以进行到更深的若干层。

470  
~  
471

```

1  /**
2   * Recursive function to find best move for computer.
3   * Returns the evaluation and sets bestMove, which
4   * ranges from 1 to 9 and indicates the best square to occupy.
5   * Possible evaluations satisfy COMP_LOSS < DRAW < COMP_WIN.
6   * Complementary function findHumanMove is Figure 10.67.
7   */
8  int TicTacToe::findCompMove( int & bestMove )
9  {
10     int i, responseValue;
11     int dc;  // dc means don't care; its value is unused
12     int value;
13
14     if( fullBoard( ) )
15         value = DRAW;
16     else
17         if( immediateCompWin( bestMove ) )
18             return COMP_WIN;  // bestMove will be set by immediateCompWin
19         else
20         {
21             value = COMP_LOSS; bestMove = 1;
22             for( i = 1; i <= 9; i++ )  // Try each square
23             {
24                 if( isEmpty( i ) )
25                 {
26                     place( i, COMP );
27                     responseValue = findHumanMove( dc );
28                     unplace( i );  // Restore board
29
30                     if( responseValue > value )
31                     {
32                         // Update best move
33                         value = responseValue;
34                         bestMove = i;
35                     }
36                 }
37             }
38         }
39     return value;
40 }

```

图10-66 极小极大三连游戏棋算法：计算机的选择

```

1 int TicTacToe::findHumanMove( int & bestMove )
2 {
3     int i, responseValue;
4     int dc; // dc means don't care; its value is unused
5     int value;
6
7     if( fullBoard( ) )
8         value = DRAW;
9     else
10    if( immediateHumanWin( bestMove ) )
11        return COMP_LOSS;
12    else
13    {
14        value = COMP_WIN; bestMove = 1;
15        for( i = 1; i <= 9; i++ ) // Try each square
16        {
17            if( isEmpty( i ) )
18            {
19                place( i, HUMAN );
20                responseValue = findCompMove( dc );
21                unplace( i ); // Restore board
22
23                if( responseValue < value )
24                {
25                    // Update best move
26                    value = responseValue;
27                    bestMove = i;
28                }
29            }
30        }
31    }
32    return value;
33 }

```

图10-67 极小极大三连游戏棋算法：人的选择

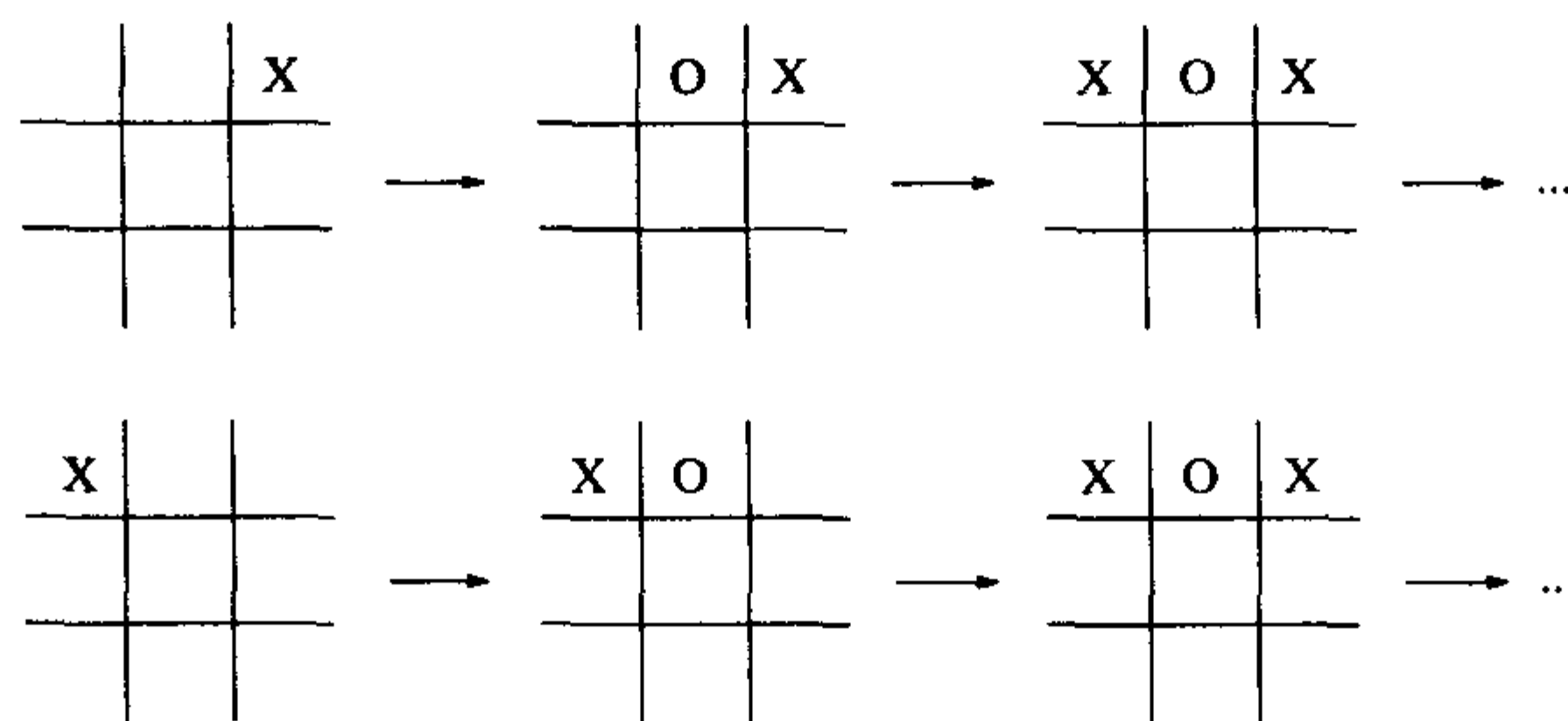


图10-68 到达同一位置的两种搜索

## 2. $\alpha$ - $\beta$ 裁剪

人们一般能够取得的最重要的改进称为 $\alpha$ - $\beta$ 裁剪 ( $\alpha$ - $\beta$  pruning)。图10-69显示了在一盘假想的棋局中用来给某个假设的位置求值的一些递归调用的迹。通常这叫作**博弈树** (game tree)。(到现在为止我们一直回避使用这个术语，因为它多少有些误导：实际上没有树是由该算法构造的，博弈树只是一个抽象的概念。) 这棵博弈树的值为44。

图10-70显示了同一棵博弈树的求值，它有一些尚未求值的结点。几乎有一半的终端结点没有被检验。下面证明计算它们的值将不改变树根的值。



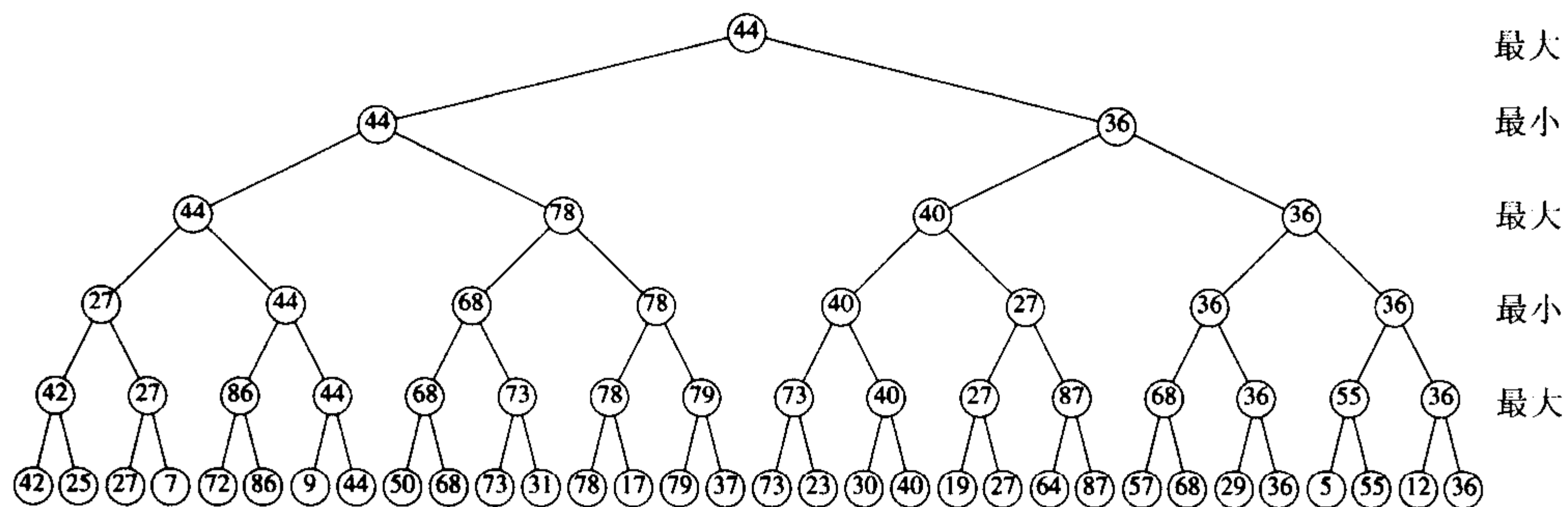


图10-69 一棵假想的博弈树

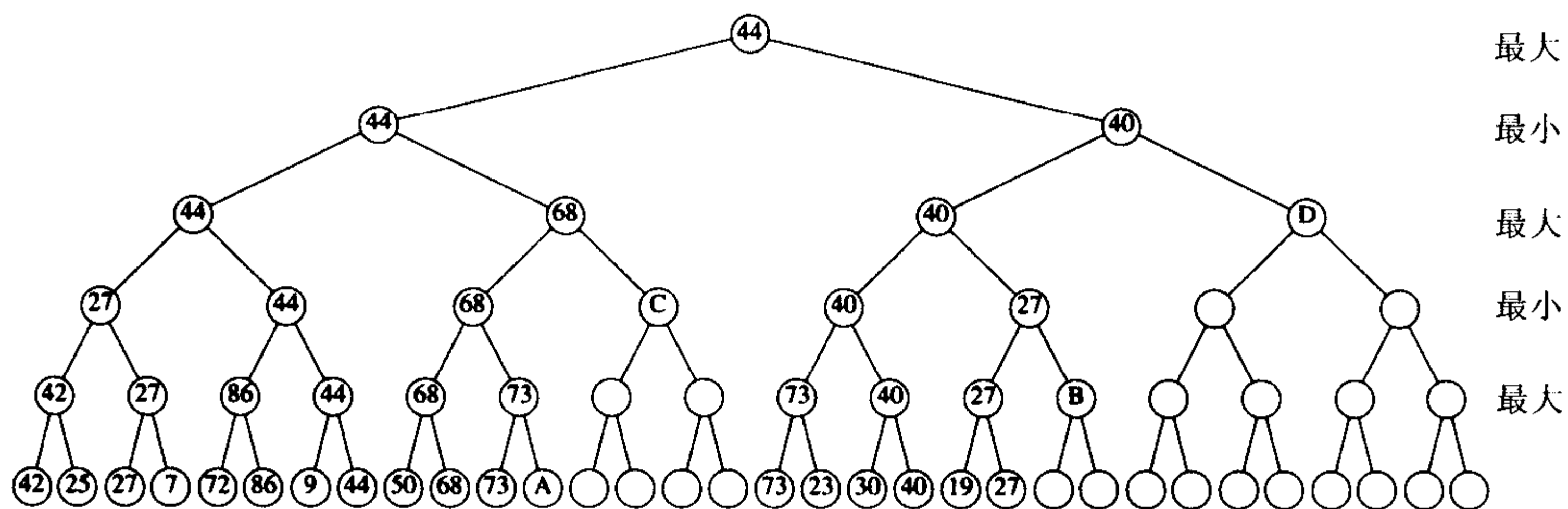


图10-70 一棵被裁剪的博弈树

首先，考虑结点 $D$ 。图10-71显示了在给 $D$ 求值时已经收集到的信息。此时，仍然处在`findHumanMove`中并正打算对 $D$ 调用`findCompMove`。然而，我们已经知道`findHumanMove`最多将返回40，因为它是一个最小结点。另一方面，它的最大结点父亲已经找到一个保证44的序列。注意， $D$ 无论如何也不可能增加这个值。因此， $D$ 不要求值。该树的这个裁剪叫作 $\alpha$ 裁剪。同样的情况出现在结点 $B$ 。为了实现 $\alpha$ 裁剪，`findCompMove`将它的尝试性的极大值（ $\alpha$ ）传递给`findHumanMove`。如果`findHumanMove`的尝试性的极小值低于这个值，那么`findHumanMove`立即返回。

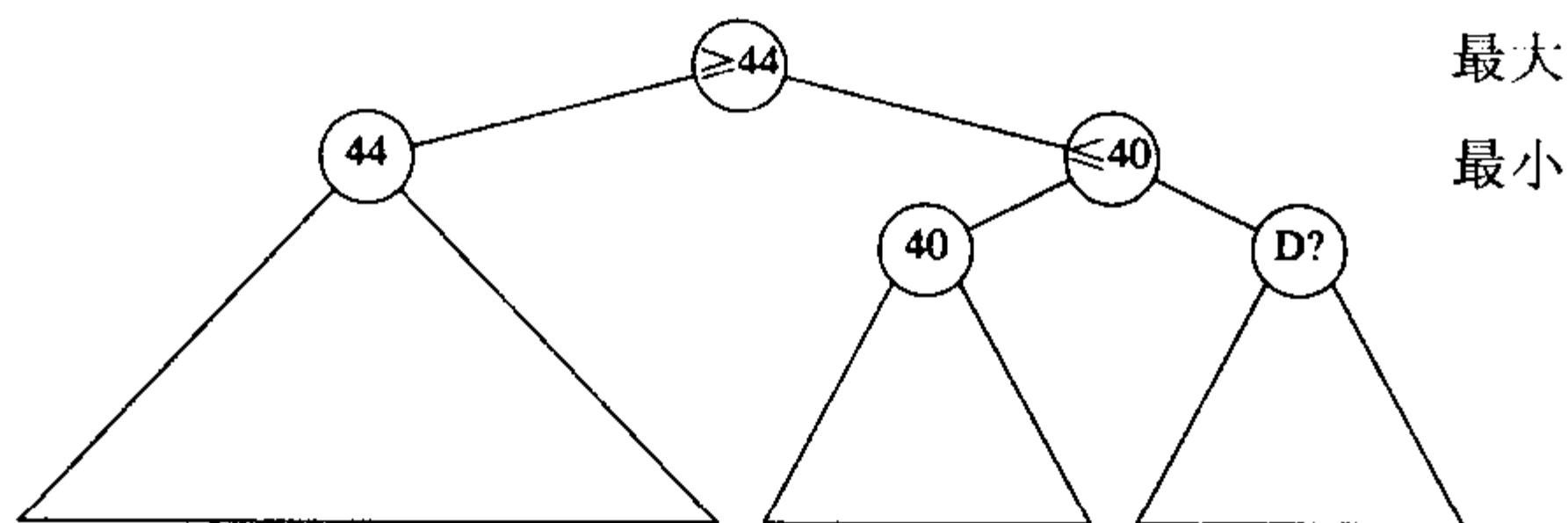


图10-71 标记“?”的结点是不重要的

473

类似的情况也发生在结点 $A$ 和 $C$ 。此时，处在`findCompMove`的中间，并且正要调用`findHumanMove`以计算 $C$ 的值。图10-72显示了结点 $C$ 的情况。不过调用了`findCompMove`的`findHumanMove`在最小层上，已经确定它能够迫使一个值最高到44（注意，对于下棋人一方低的值是好的）。由于`findCompMove`有一个尝试性的最大值68，因此 $C$ 在最小层上怎么做也不会影响到这个结果。因此， $C$ 不应该求值。这种类型的裁剪叫作 $\beta$ 裁剪，它是 $\alpha$ 裁剪的对称形式。当两种方法结合起来时得到 $\alpha$ - $\beta$ 裁剪。

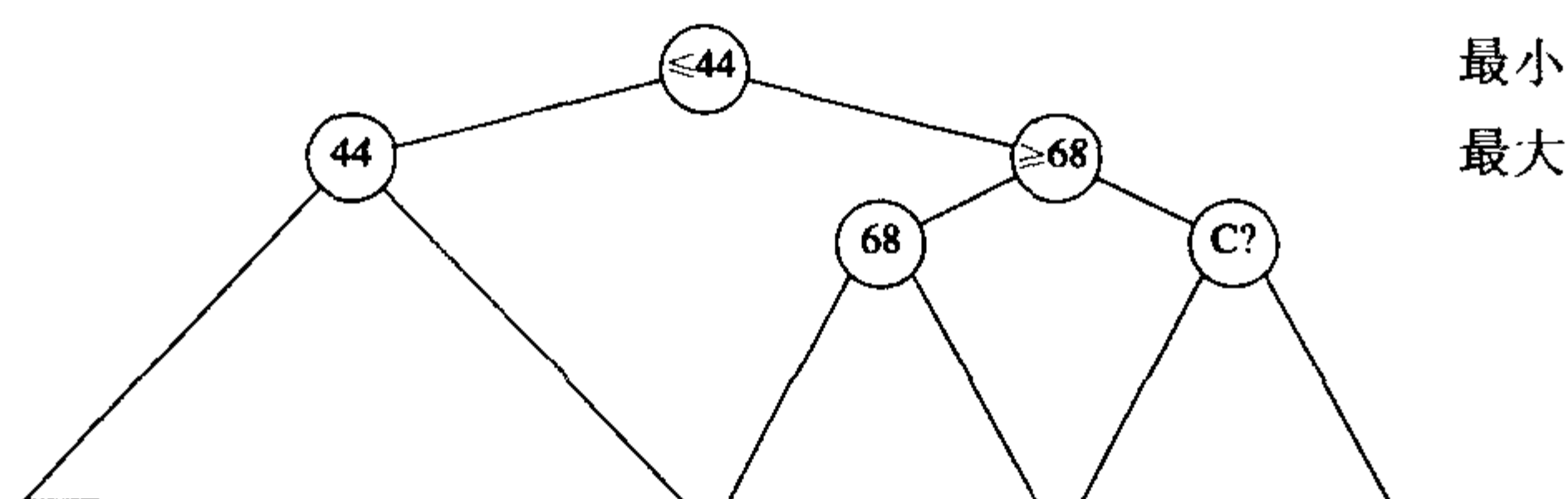


图10-72 标记“?”的结点是不重要的

实现 $\alpha$ - $\beta$ 裁剪所需的代码少得惊人。图10-73显示的是 $\alpha$ - $\beta$ 裁剪方案的一半（除去类型声明）；你不会遇到任何麻烦就能够写出另一半代码。

```

1  /**
2   * Same as before, but perform alpha-beta pruning.
3   * The main routine should make the call with
4   * alpha = COMP_LOSS and beta = COMP_WIN.
5   */
6  int TicTacToe::findCompMove( int & bestMove, int alpha, int beta )
7  {
8      int i, responseValue;
9      int dc;  // dc means don't care; its value is unused
10     int value;
11
12     if( fullBoard( ) )
13         value = DRAW;
14     else
15         if( immediateCompWin( bestMove ) )
16             return COMP_WIN;  // bestMove will be set by immediateCompWin
17         else
18         {
19             value = alpha; bestMove = 1;
20             for( i = 1; i <= 9 && value < beta; i++ )  // Try each square
21             {
22                 if( isEmpty( i ) )
23                 {
24                     place( i, COMP );
25                     responseValue = findHumanMove( dc, value, beta );
26                     unplace( i );  // Restore board
27
28                     if( responseValue > value )
29                     {
30                         // Update best move
31                         value = responseValue;
32                         bestMove = i;
33                     }
34                 }
35             }
36         }
37     return value;
38 }

```

图10-73 带有 $\alpha$ - $\beta$ 裁剪的极小极大三连游戏棋算法：计算机的选择

为了充分利用 $\alpha$ - $\beta$ 裁剪，对弈程序通常尽量对非终端结点应用求值函数，力图把最好的棋步早一些放到搜索范围内。这样的结果甚至比人们从随机顺序的结点所期望的结果还要裁剪得多。其他一些方法（如以积极的方式进行更深入的搜索）也在使用。

在实践中， $\alpha$ - $\beta$ 裁剪把搜索限制在只有 $O(\sqrt{N})$ 个结点上，这里 $N$ 是整个博弈树的大小。这是

巨大的节约，它意味着使用 $\alpha$ - $\beta$ 裁剪的搜索与非裁剪树相比能够进行到两倍的深度。前面的三连游戏棋的例子是不理想的，因为存在太多相同的值，但即使是这样，最初对97 162个结点的搜索还是被减到了4493个结点（这些计数包括非终端结点）。

在许多博弈领域，计算机跻身于世界最优秀弈者之列。所使用的技术是非常有趣的，而且可以应用到一些更严肃的问题上。更多的细节可在参考文献中找到。

## 小结

本章阐述了在算法设计中发现的5个最普通的方法。当面临一个问题的时候，花些时间考察一下这些方法能否适用是值得的。算法的适当选择，结合数据结构的审慎使用，常常能够迅速而高效地解决问题。

## 练习

- 10.1 证明：贪心算法可以将多处理器作业调度工作的平均完成时间最小化。
- 10.2 设作业 $j_1, j_2, \dots, j_N$ 为输入，其中的每一个作业都要花一个时间单位来完成。如果每个作业 $j_i$ 在时间限度 $t_i$ 内完成，那么将挣得 $d_i$ 美元，但若在时间限度以后完成则挣不到钱。
  - a. 给出一个 $O(N^2)$ 贪心算法求解该问题。
  - \*\*b.** 修改你的算法以得到 $O(M \log N)$ 的时间界。提示：时间界完全归因于将作业按照钱数排序。算法的其余部分可以使用不相交集数据结构以 $o(M \log N)$ 实现。
- 10.3 一个文件以下列频率包含冒号、空格、换行、逗号和数字：冒号（100），空格（605），换行（100），逗号（705），0（431），1（242），2（176），3（59），4（185），5（250），6（174），7（199），8（205），9（217）。构造其赫夫曼编码。
- 10.4 编码文件有一部分必须是指示赫夫曼编码的文件头。给出一种方法构建大小最多为 $O(N)$ 的文件头（除符号外），其中 $N$ 是符号的个数。
- 10.5 证明赫夫曼算法生成最优的前缀码。
- 10.6 证明：如果符号是按照频率排序的，那么赫夫曼算法可以以线性时间实现。
- 10.7 用赫夫曼算法写出一个程序实现文件压缩（和解压缩）。
- \*10.8** 证明：通过考虑下述项的序列可以迫使任意联机装箱算法至少使用 $\frac{3}{2}$ 最优箱子数： $N$ 项大小为 $\frac{1}{6} - 2\varepsilon$ ， $N$ 项大小为 $\frac{1}{3} + \varepsilon$ ， $N$ 项大小为 $\frac{1}{2} + \varepsilon$ 。
- 10.9 解释如何以时间 $O(M \log N)$ 实现首次适配算法和最佳适配算法。
- 10.10 给出在10.1.3节讨论的所有装箱方法对输入0.42, 0.25, 0.27, 0.07, 0.72, 0.86, 0.09, 0.44, 0.50, 0.68, 0.73, 0.31, 0.78, 0.17, 0.79, 0.37, 0.73, 0.23, 0.30的操作。
- 10.11 编写一个程序比较各种装箱试探方法（在时间上和所用箱子的数量上）的性能。
- 10.12 证明定理10.7。
- 10.13 证明定理10.8。
- \*10.14** 将 $N$ 个点放入一个单位方格中。证明最近点对之间的距离为 $O(N^{-1/2})$ 。
- \*10.15** 论证对于最近点算法，在带内的平均点数是 $O(\sqrt{N})$ 。提示：利用前一道练习的结果。
- 10.16 编写一个程序实现最近点对算法。
- 10.17 使用三分化中项的中项策略，快速选择算法的渐近运行时间是多少？
- 10.18 证明七分化中项的中项的快速选择算法是线性的。为什么证明中不用七分化中项的中项方法？

- 10.19 实现第7章中的快速选择算法，快速选择使用五分化中项的中项方法，并实现10.2.3节末尾的抽样算法。比较它们的运行时间。
- 10.20 许多用于计算五分化中项的中项的信息都被丢弃了。指出怎样通过更仔细地利用这些信息减少比较的次数。
- \*10.21 完成在10.2.3节末尾描述的抽样算法的分析，并解释 $\delta$ 和 $s$ 的值如何选择。
- 10.22 指出如何用递归乘法计算 $XY$ ，其中 $X = 1234$ ， $Y = 4321$ 。要包括所有的递归计算。
- 10.23 指出如何只使用三次乘法将两个复数 $X = a + bi$ 和 $Y = c + di$ 相乘。
- 10.24 a. 证明： $X_L Y_R + X_R Y_L = (X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R$ 。  
b. 它给出进行 $N$ 位数的乘法的 $O(N^{1.59})$ 算法。将该方法与书中的解法进行比较。
- 10.25 \*a. 指出如何通过求解大约为原问题三分之一大小的5个问题来完成两个数的乘法。  
\*\*b. 将该问题推广得出一个 $O(N^{1+\epsilon})$ 的算法，其中 $\epsilon$ 为大于0的任意常数。  
c. (b)问中的算法比 $O(M \log N)$ 好吗？
- 10.26 为什么Strassen算法在 $2 \times 2$ 矩阵的乘法中不使用可交换性是重要的？
- 10.27 两个 $70 \times 70$ 矩阵可以使用143 640次乘法相乘。指出这如何能够用于改进由Strassen算法给出的界。
- 10.28 计算 $A_1 A_2 A_3 A_4 A_5 A_6$ 的最优方法是什么？其中，这些矩阵的维数分别为 $10 \times 20$ ， $20 \times 1$ ， $1 \times 40$ ， $40 \times 5$ ， $5 \times 30$ ， $30 \times 15$ 。
- 10.29 证明下列贪心算法均不能进行链式矩阵乘法。在每一步  
a. 计算最节省的乘法。  
b. 计算最昂贵的乘法。  
c. 计算两个矩阵 $M_i$ 和 $M_{i+1}$ 之间的乘法，使得在 $M_i$ 中的列数最小（使用上面法则之一）。
- 10.30 编写一个程序计算矩阵乘法的最佳顺序。注意，要包括显示具体顺序的例程。
- 10.31 指出下列单词的最优二叉查找树，其中括号内是单词出现的频率： $a(0.18)$ ， $and(0.19)$ ， $I(0.23)$ ， $it(0.21)$ ， $or(0.19)$ 。
- \*10.32 将最优二叉查找树算法扩展到可以对不成功的搜索进行。在这种情况下， $q_j$ 是对任意满足 $w_j < W < w_{j+1}$ 的单词 $W$ 执行一次查找的概率，其中 $1 \leq j < N$ 。 $q_0$ 是对 $W < w_1$ 的单词 $W$ 执行一次查找的概率，而 $q_N$ 是对 $W > w_N$ 执行一次查找的概率。注意， $\sum_{i=1}^N p_i + \sum_{j=0}^N q_j = 1$ 。
- \*10.33 设 $C_{i,i} = 0$ ，此外

$$C_{i,j} = W_{i,j} + \min_{i < k \leq j} (C_{i,k-1} + C_{k,j})$$

设 $W$ 满足四边形不等式 (quadrangle inequality)，即对所有的 $i \leq i' \leq j \leq j'$ ，有

$$W_{i,j} + W_{i',j'} \leq W_{i',j} + W_{i,j'}$$

进一步假设 $W$ 是单调的：如果 $i \leq i'$ 及 $j \leq j'$ ，那么 $W_{i,j} \leq W_{i',j'}$ 。

- a. 证明 $C$ 满足四边形不等式。
- b. 令 $R_{i,j}$ 是使 $C_{i,k-1} + C_{k,j}$ 达到最小值的最大的 $k$ （也就是说，在相同的情形下选择最大的 $k$ ）。证明： $R_{i,j} \leq R_{i,j+1} \leq R_{i+1,j+1}$
- c. 证明 $R$ 沿着每一行和列是非减的。
- d. 证明 $C$ 中所有的项可以以 $O(N^2)$ 时间计算。
- e. 使用这些技巧以 $O(N^2)$ 可以解决哪个动态规划算法？
- 10.34 编写一个例程从10.3.4节中的算法重新构造最短路径。
- 10.35 二项式系数 $C(N, k)$ 可以递归定义如下： $C(N, 0) = 1$ ， $C(N, N) = 1$ ，且对于 $0 < k < N$ ， $C(N, k) = C(N-1, k) + C(N-1, k-1)$ 。编写一个函数并给出如下计算二项式系数的运行时间的分析：  
a. 递归计算。  
b. 使用动态规划算法。
- 10.36 编写在跳跃表中执行插入、删除以及查找的例程。



10.37 给出跳跃表操作的期望时间为 $O(\log N)$ 的形式化证明。

10.38 图10-74显示了抛一枚硬币的例程，假设random返回一个整数（这在许多系统中常见）。如果随机数生成器使用形如 $M=2^B$ 的模（遗憾的是这在许多系统上流行），那么跳跃表算法的期望性能如何？

```

1 CoinSide flip( )
2 {
3     if( ( random( ) % 2 ) == 0 )
4         return HEADS;
5     else
6         return TAILS;
7 }

```

图10-74 有问题的抛币器

10.39 a. 用取幂算法证明 $2^{340} \equiv 1 \pmod{341}$ 。

b. 指出随机化素性测试当 $N=561$ 时对于 $A$ 的多种选择是如何工作的。

10.40 实现公路收费点重建算法。

10.41 如果两个点集产生相同的距离集合而不互相转换，那么这两个点集称为是同度的（homometric）。下列距离集合给出两个不同的点集： $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16, 17\}$ 。求出这两个点集。

10.42 扩展重建算法使给定一个距离集合找出所有的同度点集。

10.43 指出图10-75中的树的 $\alpha$ - $\beta$ 裁剪结果。

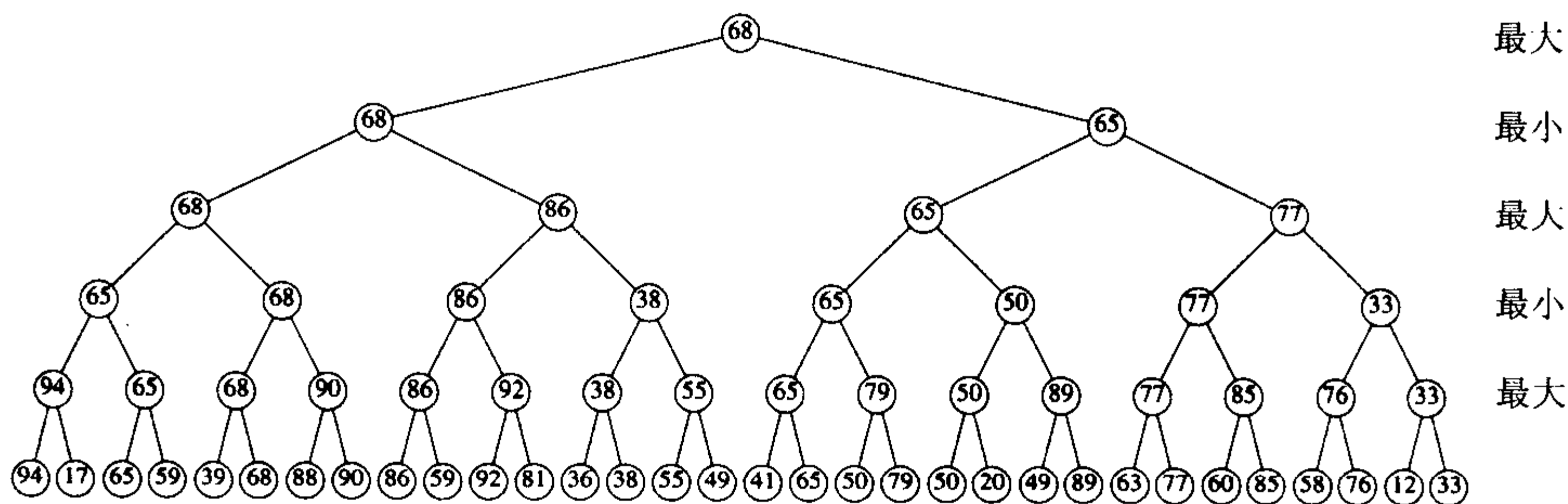


图10-75 博弈树，该树可以裁剪

10.44 a. 图10-73中的程序实现 $\alpha$ 裁剪还是 $\beta$ 裁剪？

b. 实现与其互补的例程。

10.45 写出三连游戏棋剩下的过程。

10.46 一维装圆问题(one-dimensional circle packing problem)描述如下：有 $N$ 个半径分别是 $r_1, r_2, \dots, r_N$ 的圆。将这些圆装到一个盒子中使得每个圆都与盒子的底边相切，并且圆按原来的顺序排列。该问题是找出最小尺寸的盒子的宽度。图10-76显示了一个例子，圆的半径分别为2、1、2。最小尺寸盒子的宽度为 $4 + 4\sqrt{2}$ 。

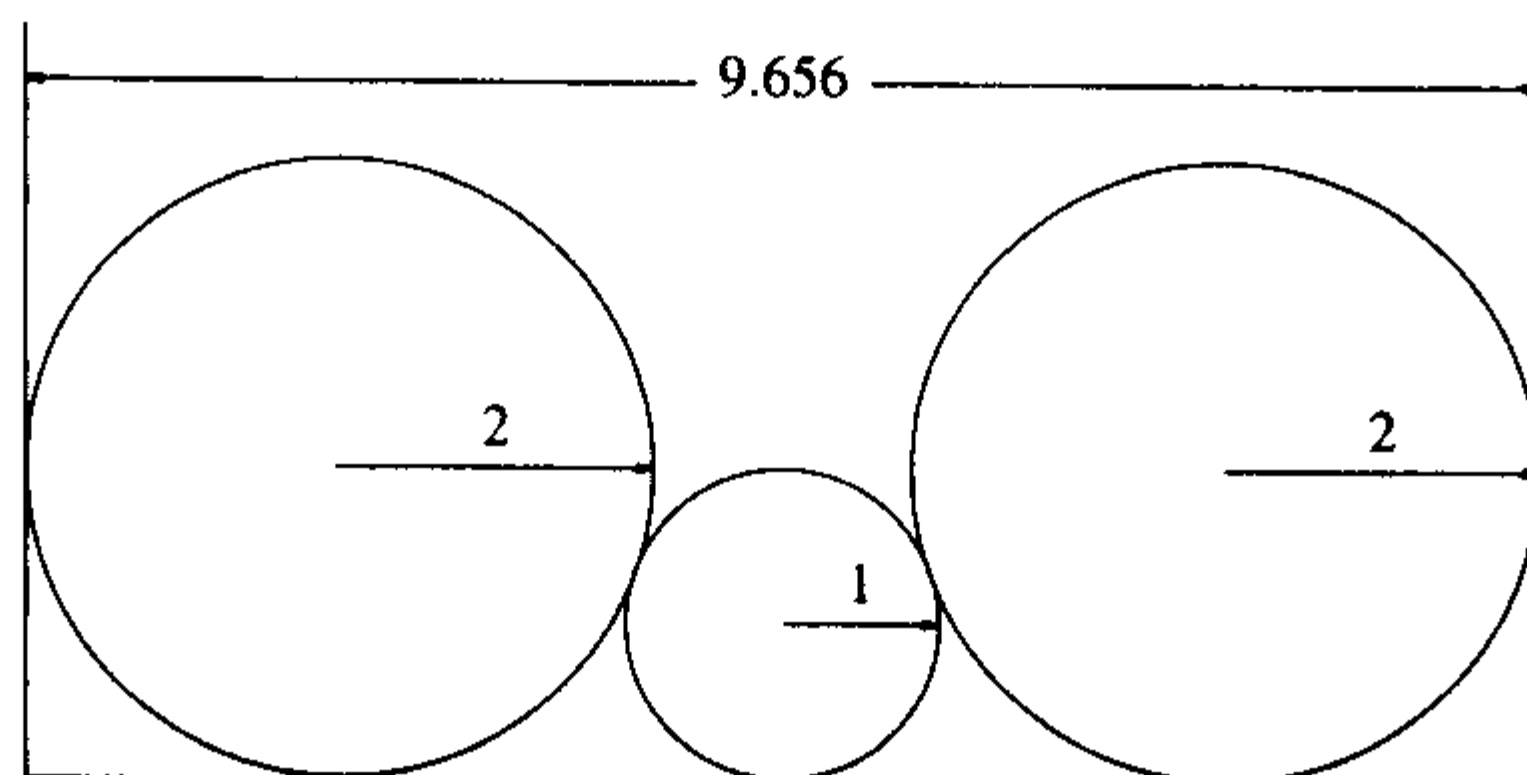


图10-76 装圆问题示例

- \*10.47 设无向图 $G$ 的边满足三角形不等式： $c_{u,v} + c_{v,w} \geq c_{u,w}$ 。指出如何计算值最多为最优路径两倍的旅行商环游。提示：构造最小生成树。
- \*10.48 假设你是邀请赛的主管，需要安排 $N = 2^k$ 个运动员间的循环赛。在这次邀请赛上，每人每天恰好打一场比赛； $N-1$ 天后，每对选手间均已进行了比赛。给出一个递归算法安排比赛。
- 10.49 \*a. 证明：在循环赛中，总能够以顺序 $p_1, p_2, \dots, p_N$ 安排运动员，使得对所有 $1 \leq j < N$ ， $p_{i_j}$ 赢得对 $p_{i_{j+1}}$ 的比赛。
- b. 给出一个 $O(M \log N)$ 算法来找出一个这样的安排。你的算法可以作为上一问(a)的证明。
- \*10.50 给定平面上 $N$ 个点的集合 $P = p_1, p_2, \dots, p_N$ 。Voronoi图是将平面分成 $N$ 个区域 $R_i$ ，使得 $R_i$ 中所有的点比 $P$ 中任何其他的点都更接近 $p_i$ 。图10-77显示了7个(细心安排的)点的Voronoi图。给出一个 $O(M \log N)$ 算法构造Voronoi图。
- \*10.51 凸多边形(convex polygon)是具有如下性质的多边形：端点位于多边形上的任意线段全部落在该多边形中。凸包(convex hull)问题是找出一个将平面上的点集围住的(面积)最小的凸多边形。图10-78显示了40个点的点集的凸包。给出找出凸包的一个 $O(M \log N)$ 算法。

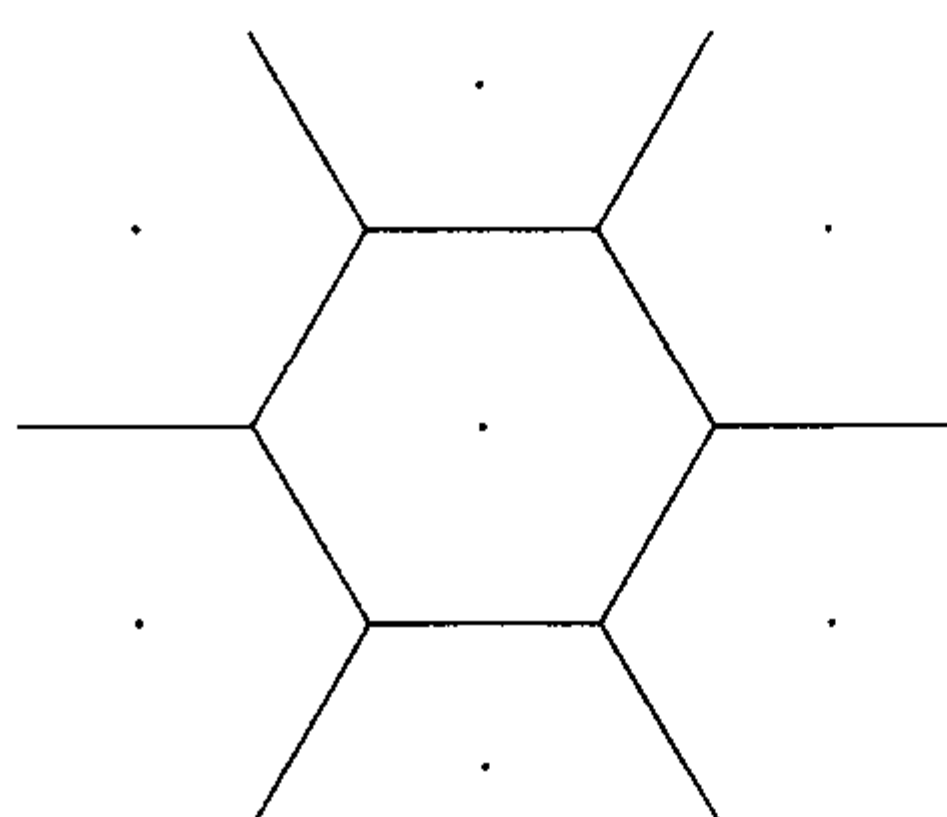


图10-77 Voronoi图

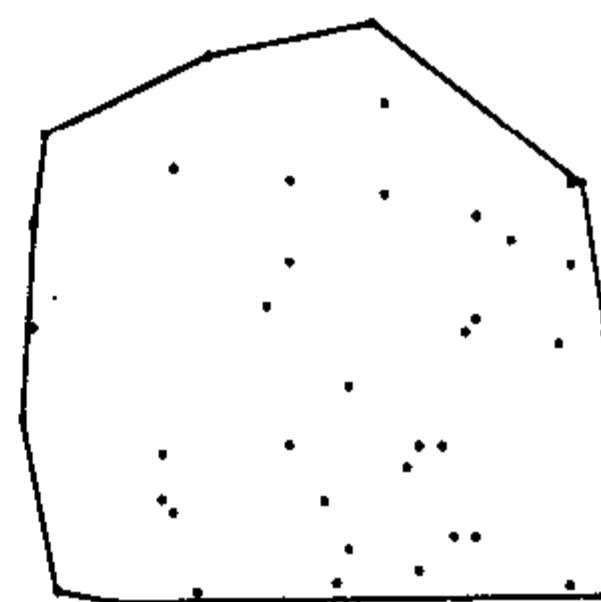


图10-78 一个凸包的例子

- \*10.52 考虑正确调整一个段落的问题。段落由一系列长度分别为 $a_1, a_2, \dots, a_N$ 的单词 $w_1, w_2, \dots, w_N$ 组成，我们希望把它断成长度为 $L$ 的一些行。单词间由空白分隔，空白的理想长度是 $b$  (mm)，但是空白在必要的时候可以伸长或收缩(不过必须大于0)，使得一行 $w_i w_{i+1} \dots w_j$ 的长度恰好是 $L$ 。然而，对于每一个空白 $b'$ 我们要装填 $|b' - b|$ 个丑点(ugliness point)。不过，最后一行例外，我们只在 $b' < b$ 的时候装填(换句话说，装填只在收缩的时候进行)，因为最后一行不需要调整。这样，如果 $b_i$ 是 $a_i$ 和 $a_{i+1}$ 之间的空白的长度，那么任何一行(最后一行除外) $w_i w_{i+1} \dots w_j (j > i)$ 的丑点设置为 $\sum_{k=i}^{j-1} |b_k - b| = (j - i)|b' - b|$ ，其中 $b'$ 是该行上空白的平均大小。这只在 $b' < b$ 时对最后一行适用，否则，最后一行根本不必装填丑点。
- a. 给出一个动态规划算法来找出将 $w_1, w_2, \dots, w_N$ 排成长度为 $L$ 的一些行的最少的丑点设置。提示：对于 $i = N, N-1, \dots, 1$ ，计算 $w_i, w_{i+1}, \dots, w_N$ 的最好的排版方式。
- b. 给出你的算法的时间和空间复杂度(作为单词个数 $N$ 的函数)。
- c. 考虑使用固定宽度字体的特殊情况，假设 $b$ 的最优值为1(空格)。在这种情况下，不允许空白收缩，因为下一个最小的空白空间是0。给出一个线性时间算法生成在这种情形下的最少的丑点设置。
- \*10.53 最长递增子序列(longest increasing subsequence)问题如下：给定数 $a_1, a_2, \dots, a_N$ ，找出使得 $a_{i_1} < a_{i_2} < \dots < a_{i_k}$ 且 $i_1 < i_2 < \dots < i_k$ 的最大的 $k$ 值。作为一个例子，如果输入为3, 1, 4, 1, 5, 9, 2, 6, 5，那么最大递增子序列的长度为4(该子列为1, 4, 5, 9)。给出一个 $O(N^2)$ 算法求解最长递增子序列问题。
- \*10.54 最长公共子序列(longest common subsequence)问题如下：给定两个序列 $A = a_1, a_2, \dots, a_M$ 和 $B = b_1, b_2, \dots, b_N$ ，找出 $A$ 和 $B$ 二者共有的最长子序列 $C = c_1, c_2, \dots, c_k$ 的长度 $k$ 。例如，若

$$A = d, y, n, a, m, i, c$$

和

$$B = p, r, o, g, r, a, m, m, i, n, g$$

则最长公共子序列为  $a, m, i$ ，其长度为3。给出一个算法求解最长公共子序列问题，算法应该以  $O(MN)$  时间运行。

\*10.55 字型匹配问题 (pattern matching problem) 如下：给定一个文本串  $S$  和一种字型  $P$ ，找出  $P$  在  $S$  中的首次出现。近似字型匹配 (approximate pattern matching) 允许以下三种类型的  $k$  次误匹配：

- (1) 一个字符在  $S$  中但不在  $P$  中。
- (2) 一个字符在  $P$  中但不在  $S$  中。
- (3)  $P$  和  $S$  可以在一个位置上不同。

例如，若在串 “data structures txtbork” 中搜索 “textbook” 最多允许三次误匹配，则找到一个匹配（插入一个  $e$ ，将一个  $r$  改变成  $o$ ，删除一个  $p$ ）。给出一个  $O(MN)$  算法求解近似串匹配问题，其中  $M=|P|$  以及  $N=|S|$ 。

\*10.56 背包问题 (knapsack problem) 的一种形式如下：给定整数集合  $A = a_1, a_2, \dots, a_N$  和一个整数  $K$ 。存在  $A$  的一个子集其和恰好为  $K$  吗？

- a. 给出一个算法以时间  $O(NK)$  求解背包问题。
- b. 为什么它不证明  $P = NP$ ？

\*10.57 给你一个货币系统，它的硬币值  $c_1, c_2, \dots, c_N$  分（以递减顺序排列）。

- a. 给出一个算法计算找  $K$  分零钱所需的最少硬币数。
- b. 给出一个算法计算找  $K$  分零钱的不同方法数。

\*10.58 考虑将8个皇后放到一张（8行8列的）棋盘上的问题。如果两个皇后处在同一行、同一列或同一条（不必是主）对角线上，则说她们是互相对攻的。

- a. 给出一个随机化算法把8个非攻击皇后放到棋盘上。
- b. 给出一个回溯算法解决同一问题。
- c. 实现这两个算法并比较它们的运行时间。

\*10.59 在国际象棋中，在  $R$  行  $C$  列上的国王可以走到  $1 \leq R' \leq B$  行和  $1 \leq C' \leq B$  列（其中  $B$  是棋盘的大小）处，假设或者  $|R - R'| = 2$  且  $|C - C'| = 1$ ，或者  $|R - R'| = 1$  且  $|C - C'| = 2$ 。

一次马的环游是马在棋盘上的一系列跳行，它恰好访问所有的方格一次而且最后又回到开始的位置。

- a. 如果  $B$  是奇数，证明马的环游不存在。
- b. 给出一个回溯算法找出一次马的环游。

10.60 考虑图10-79中的递归算法，该算法在一个无环图中寻找从  $S$  到  $T$  的最短加权路径。

```
Distance Graph::shortest( s, t )
{
    Distance dt, temp;

    if( s == t )
        return 0;

    dt = ∞;
    for each Vertex v adjacent to s
    {
        tmp = shortest( v, t );
        if( cs,v + tmp < dt )
            dt = cs,v + tmp;
    }
    return dt;
}
```

图10-79 递归的最短路径算法

- a. 这个算法对于一般的图为什么行不通?
- b. 证明该算法对无环图可以终止。
- c. 该算法的最坏情形运行时间是多少?

10.61 令 $A$ 为元素是0和1的 $N$ 行 $N$ 列矩阵。 $A$ 的子矩阵 $S$ 由形成方阵的任意一组相邻项组成。

- a. 设计一种 $O(N^2)$ 算法, 确定 $A$ 中1的最大子矩阵的阶数。例如, 在下列矩阵中, 这种最大的子矩阵是4行4列的方阵。

```

10111000
00010100
00111000
00111010
00111111
01011110
01011110
00011110

```

**\*\*b.** 如果 $S$ 可以不只是方形而且还可以是矩形, 重复(a)问的设计。最大的含义是由面积来度量的。

10.62 即使计算机有一步就能立即赢棋的棋步, 若它检测到另外一步保证赢棋的棋, 则它也可能不走立即赢棋的棋步。一些早期的国际象棋程序在这一点上是有问题的, 当检测到被迫的赢招时, 它们陷入重复位置上的循环, 因此允许对方宣布和棋。在三连游戏棋中没有这个问题, 因为程序最终将赢棋。修改三连游戏棋算法, 使得当找到赢棋位置时, 导致最快赢棋的棋步总会被采纳。做法是: 通过把9-depth加到COMP\_WIN使得最快的赢棋位置给出最高的值。

10.63 编写一个程序对弈5行5列的五连游戏棋, 其中4个在一行则赢棋。你能搜索到终端结点吗?

10.64 Boggle游戏由字母的网格和一个单词表组成。游戏的目标是找出网格中的一些单词, 它们满足条件: 两个相邻的字母必须在网格中也相邻, 并且每个单词最多使用一次网格中的每一项。编写进行Boggle游戏的程序。

10.65 编写进行MAXIT游戏的程序。棋盘是 $N$ 行 $N$ 列的网格, 游戏开始时这些网格随机放入整数。指定一个位置为初始当前位置, 游戏双方交替行棋。每次, 行棋的一方必须在当前的行或列上选取一个网格元素, 所选位置的值则被加到该方的得分中去, 并且这个位置变成了当前位置而不能再选用。游戏双方轮流行棋直到当前行和列上的网格元素都被选过, 此时游戏终止, 得到高分的人获胜。

10.66 奥赛罗棋在6行6列的棋盘上进行, 而且总是黑方赢棋。编写一个程序证明之。如果双方都玩到最优, 那么最后的得分是多少?

483

484

## 参考文献

关于赫夫曼编码的原始论文为[23]。该算法的各种变形在[31]、[34]和[35]中讨论。另一种流行的压缩方案是Ziv-Lempel编码[62]和[63], 这里的编码具有固定的长度, 它们代表串而不是字符。[9]和[37]是对普通压缩方案的优秀的综述。

装箱问题探测法的分析最初出现在Johnson的博士论文中, 并在[24]中发表。在练习10.8中给出的联机装箱问题改进的下界来自论文[59]; 这个结果在[38]和[57]中得到进一步的改进。[51]则描述了对联机装箱问题的另一种处理方法。

定理10.7摘自文献[8]。最近点算法出自[52]。[54]描述了公路收费点重建问题及其应用。指数最坏情形的输入由[61]给出。在相对新的计算几何领域中的两本老书是[15]和[47]。[43]和[44]则包含一些更新的成果。[2]包含了在麻省理工学院所教计算几何课程的讲稿, 它包括一个广泛的文献目录。

线性时间选择算法出自[10]。[18]讨论以 $1.5 N$ 次期望比较找出中位数的取样方法。 $O(N^{1.59})$ 的乘法来自



[25], 在[11]和[27]中讨论了若干推广。Strassen算法出自短文[55], 这篇论文只叙述一些结果, 此外没有太多的内容。Pan[45]给出了若干分治算法, 包括练习10.27中的算法。已知最好的界是 $O(N^{2.376})$ , 该结果归因于Coppersmith和Winograd[13]。

动态规划的经典文献是著作[6]和[7]。矩阵排序问题最初在[20]中研究, 论文[22]证明该问题可以以 $O(M \log N)$ 时间求解。

Knuth[27]提供了一个 $O(N^2)$ 算法构建最优二叉查找树。所有点对的最短路径算法出自Floyd[17]。理论上更好的 $O(N^3(\log \log N / \log N)^{1/3})$ 算法由Fredman[19]给出, 不过它并不实用, 这不奇怪。稍微改进的界(指数为1/2而不是1/3)在[56]中给出; 相关的结果也见于[4]。在某些条件下, 动态规划的运行时间可以自动地改进 $N$ 的一个因子或更多, 这在练习10.33、论文[16]和[60]中都有讨论。

485

随机数生成器的讨论基于[46]。Park和Miller把轻便的实现方法归因于Schrage[53]。跳跃表由Pugh在[48]中讨论, 另一种结构即treap树在第12章讨论。随机化素性测试算法属于Miller[40]和Rabin[50]。 $A$ 的最多 $(N-9)/4$ 个值将会使算法失误的定理源于Monier[41]。另外一些随机化算法在[49]中讨论。随机化技巧的更多的例子可在[22]、[26]和[42]中找到。

关于 $\alpha$ - $\beta$ 裁剪更多的信息可以查阅[1]、[29]和[32]。一些下国际象棋、西洋跳棋、奥赛罗棋以及十五子棋的顶尖级的程序均已达到世界级水平。[36]描述了一个奥赛罗棋的程序, 这篇论文出自计算机游戏(大部分是国际象棋)专刊; 这个专刊是思想的金矿。其中有一篇论文描述当棋盘上只有少数棋子的时候使用动态规划彻底解决残局的下法。相关研究已经导致在某些情况下50步规则的改变。

练习10.41在[9]中解决。确定没有重复距离的同度点集对于 $N > 6$ 是否存在是一个尚未解决的问题。Christofides[13]给出了练习10.47的一种解法, 此外还给出了一个最多以 $\frac{3}{2}$ 倍的最优时间生成一个环游的算法。练习10.52在[30]中讨论。练习10.55在[58]中解决。在[33]中给出一个 $O(kN)$ 算法。练习10.57在[12]中讨论, 但不要被论文的标题所误导。

1. B. Abramson, "Control Strategies for Two-Player Games," *ACM Computing Surveys*, 21 (1989), 137-161.
2. A. Aggarwal and J. Wein, *Computational Geometry: Lecture Notes for 18.409*, MIT Laboratory for Computer Science, 1988.
3. M. Agrawal, N. Kayal, and N. Saxena, "Primes in P (preprint)" (2002), (see <http://www.cse.iitk.ac.in/news/primalty.pdf>)
4. N. Alon, Z. Galil, and O. Margalit, "On the Exponent of the All-Pairs Shortest Path Problem," *Proceedings of the Thirty-Second Annual Symposium on the Foundations of Computer Science* (1991), 569-575.
5. T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for Text Compression," *ACM Computing Surveys*, 21 (1989), 557-591.
6. R. E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, N.J., 1957.
7. R. E. Bellman and S. E. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, N.J., 1962.
8. J. L. Bentley, D. Haken, and J. B. Saxe, "A General Method for Solving Divide-and-Conquer Recurrences," *SIGACT News*, 12 (1980), 36-44.
9. G. S. Bloom, "A Counterexample to the Theorem of Piccard," *Journal of Combinatorial Theory A* (1977), 378-379.
10. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time Bounds for Selection," *Journal of Computer and System Sciences*, 7 (1973), 448-461.
11. A. Borodin and J. I. Munro, *The Computational Complexity of Algebraic and Numerical Problems*, American Elsevier, New York, 1975.
12. L. Chang and J. Korsh, "Canonical Coin Changing and Greedy Solutions," *Journal of the ACM*, 23 (1976), 418-422.
13. N. Christofides, "Worst-case Analysis of a New Heuristic for the Traveling Salesman Problem,"

486

- Management Science Research Report #388*, Carnegie-Mellon University, Pittsburgh, PA, 1976.
14. D. Coppersmith and S. Winograd, "Matrix Multiplication via Arithmetic Progressions," *Proceedings of the Nineteenth Annual ACM Symposium on the Theory of Computing* (1987), 1-6.
  15. H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, 1987.
  16. D. Eppstein, Z. Galil, and R. Giancarlo, "Speeding up Dynamic Programming," *Proceedings of the Twenty-ninth Annual IEEE Symposium on the Foundations of Computer Science* (1988), 488-495.
  17. R. W. Floyd, "Algorithm 97: Shortest Path," *Communications of the ACM*, 5 (1962), 345.
  18. R. W. Floyd and R. L. Rivest, "Expected Time Bounds for Selection," *Communications of the ACM*, 18 (1975), 165-172.
  19. M. L. Fredman, "New Bounds on the Complexity of the Shortest Path Problem," *SIAM Journal on Computing*, 5 (1976), 83-89.
  20. S. Godbole, "On Efficient Computation of Matrix Chain Products," *IEEE Transactions on Computers*, 9 (1973), 864-866.
  21. R. Gupta, S. A. Smolka, and S. Bhaskar, "On Randomization in Sequential and Distributed Algorithms," *ACM Computing Surveys*, 26 (1994), 7-86.
  22. T. C. Hu and M. R. Shing, "Computations of Matrix Chain Products, Part I," *SIAM Journal on Computing*, 11 (1982), 362-373.
  23. D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE*, 40 (1952), 1098-1101.
  24. D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case Performance Bounds for Simple One-Dimensional Packing Algorithms," *SIAM Journal on Computing*, 3 (1974), 299-325.
  25. A. Karatsuba and Y. Ofman, "Multiplication of Multi-digit Numbers on Automata," *Doklady Akademii Nauk SSSR*, 145 (1962), 293-294.
  26. D. R. Karger, "Random Sampling in Graph Optimization Problems," Ph.D. thesis, Stanford University, 1995.
  27. D. E. Knuth, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*, 3d ed., Addison-Wesley, Reading, Mass., 1998.
  28. D. E. Knuth, "Optimum Binary Search Trees," *Acta Informatica*, 1 (1971), 14-25.
  29. D. E. Knuth, "An Analysis of Alpha-Beta Cutoffs," *Artificial Intelligence*, 6 (1975), 293-326.
  30. D. E. Knuth, *TEX and Metafont, New Directions in Typesetting*, Digital Press, Bedford, Mass., 1981.
  31. D. E. Knuth, "Dynamic Huffman Coding," *Journal of Algorithms*, 6 (1985), 163-180.
  32. D. E. Knuth and R. W. Moore, "Estimating the Efficiency of Backtrack Programs," *Mathematics of Computation*, 29 (1975), 121-136.
  33. G. M. Landau and U. Vishkin, "Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm," *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing* (1986), 220-230.
  34. L. L. Larmore, "Height-Restricted Optimal Binary Trees," *SIAM Journal on Computing*, 16 (1987), 1115-1123.
  35. L. L. Larmore and D. S. Hirschberg, "A Fast Algorithm for Optimal Length-Limited Huffman Codes," *Journal of the ACM*, 37 (1990), 464-473.
  36. K. Lee and S. Mahajan, "The Development of a World Class Othello Program," *Artificial Intelligence*, 43 (1990), 21-36.
  37. D. A. Lelewer and D. S. Hirschberg, "Data Compression," *ACM Computing Surveys*, 19 (1987), 261-296.
  38. H. W. Lenstra, Jr. and C. Pomerance, "Primality Testing with Gaussian Periods," Manuscript (2003).
  39. E. M. Liang, "A Lower Bound for On-line Bin Packing," *Information Processing Letters*, 10 (1980), 76-79.
  40. G. L. Miller, "Riemann's Hypothesis and Tests for Primality," *Journal of Computer and System Sciences*, 13 (1976), 300-317.
  41. L. Monier, "Evaluation and Comparison of Two Efficient Probabilistic Primality Testing Algorithms," *Theoretical Computer Science*, 12 (1980), 97-108.

42. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York (1995).
43. K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice-Hall, Englewood Cliffs, N.J. (1994).
44. J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, New York (1994).
45. V. Pan, "Strassen's Algorithm Is Not Optimal," *Proceedings of the Nineteenth Annual IEEE Symposium on the Foundations of Computer Science* (1978), 166-176.
46. S. K. Park and K. W. Miller, "Random Number Generators: Good Ones Are Hard To Find," *Communications of the ACM*, 31 (1988), 1192-1201. (See also *Technical Correspondence*, in 36 (1993), 105-110.)
47. E. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.
48. W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM*, 33 (1990), 668-676.
49. M. O. Rabin, "Probabilistic Algorithms," in *Algorithms and Complexity, Recent Results and New Directions* (J. F. Traub, ed.), Academic Press, New York, 1976, 21-39.
50. M. O. Rabin, "Probabilistic Algorithms for Testing Primality," *Journal of Number Theory*, 12 (1980), 128-138.
51. P. Ramanan, D. J. Brown, C. C. Lee, and D. T. Lee, "On-line Bin Packing in Linear Time," *Journal of Algorithms*, 10 (1989), 305-326.
52. M. I. Shamos and D. Hoey, "Closest-Point Problems," *Proceedings of the Sixteenth Annual IEEE Symposium on the Foundations of Computer Science* (1975), 151-162.
53. L. Schrage, "A More Portable FORTRAN Random Number Generator," *ACM Transactions on Mathematics Software*, 5 (1979), 132-138.
54. S. S. Skiena, W. D. Smith, and P. Lemke, "Reconstructing Sets from Interpoint Distances," *Proceedings of the Sixth Annual ACM Symposium on Computational Geometry* (1990), 332-339.
55. V. Strassen, "Gaussian Elimination Is Not Optimal," *Numerische Mathematik*, 13 (1969), 354-356.
56. T. Takaoka, "A New Upper Bound on the Complexity of the All-Pairs Shortest Path Problem," *Information Processing Letters*, 43 (1992), 195-199.
57. A. van Vliet, "An Improved Lower Bound for On-Line Bin Packing Algorithms," *Information Processing Letters*, 43 (1992), 277-284.
58. R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM*, 21 (1974), 168-173.
59. A. C. Yao, "New Algorithms for Bin Packing," *Journal of the ACM*, 27 (1980), 207-227.
60. F. F. Yao, "Efficient Dynamic Programming Using Quadrangle Inequalities," *Proceedings of the Twelfth Annual ACM Symposium on the Theory of Computing* (1980), 429-435.
61. Z. Zhang, "An Exponential Example for a Partial Digest Mapping Algorithm," *Journal of Computational Molecular Biology*, 1 (1994), 235-239.
62. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory* IT23 (1977), 337-343.
63. J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-rate Coding," *IEEE Transactions on Information Theory* IT24 (1978), 530-536.



## 摊还分析

**本**章将对第4章和第6章出现的几种高级数据结构的运行时间进行分析，特别是将考虑任意顺序的 $M$ 次操作的最坏情形运行时间。这与更一般的分析有所不同，后者是对任意单次的操作给出最坏情形的时间界。

例如，我们已经看到AVL树以每次操作 $O(\log N)$ 最坏情形时间支持标准的树操作。AVL树在实现上多少有些复杂，这不仅是因为存在许多不同的情况，还因为高度平衡信息必须保存和正确地更新。使用AVL树的原因在于，对非平衡查找树的一系列 $\Theta(N)$ 操作可能需要 $\Theta(N^2)$ 时间，从而导致高昂的消耗。对于查找树来说，一次操作的 $O(N)$ 最坏情形运行时间并不是真正的问题，主要的问题是这种情形可能反复发生。伸展树提供了一种较好的方法，虽然任意操作仍然需要 $\Theta(N)$ 时间，但是这种退化行为不可能反复发生，而且可以证明，任意顺序的 $M$ 次操作（总共）花费 $O(M \log N)$ 最坏情形时间。因此，在长时期运行中这种数据结构的行为就像是每次操作花费 $O(\log N)$ 时间一样。我们把它称为**摊还时间界**（amortized time bound）。

摊还界比对应的最坏情形界弱，因为它对任意单次操作提供不了保障。由于这个问题一般来说并不重要，因此如果能够对一系列操作保持相同的界同时又简化数据结构，那么我们愿意牺牲单次操作的界。摊还界比相同的平均情形界要强。例如，二叉查找树每次操作的平均时间为 $O(\log N)$ ，但是对于连续 $M$ 次操作仍然可能花费 $O(MN)$ 时间。

因为得到摊还界需要查看整个操作序列而不仅仅是一次操作，所以我们希望分析更具技巧性。我们将看到这种期望一般会实现。

本章中，我们将：

- 分析二项队列操作。
- 分析斜堆。
- 介绍并分析斐波那契堆。
- 分析伸展树。

491

### 11.1 一个无关的智力问题

考虑下列问题：将两只小猫放在足球场的两端，相距100码。它们以每分钟10码的速度相向行走。同时，这两只小猫的母亲在足球场的一端，它以每分钟100码的速度跑步。猫妈妈从一只小猫跑到另一只小猫，来回轮流跑而速度不减，一直跑到两只小猫（以及猫妈妈）在中场相遇。问猫妈妈跑了多远？

使用蛮力计算不难解决这个问题。具体细节留给读者，不过，预计这个计算将涉及计算无穷几何级数的和。虽然这种直接计算能够得到答案，但是实际上通过引入一个附加变量，即时间，可以得到简单得多的解法。



因为两只小猫相距100码远而且以每分钟20码的合速度互相接近，所以它们花5分钟即可到达中场。由于猫妈妈每分钟跑100码，因此它跑的总距离是500码。

这个问题阐述了一个思路，即有时候间接求解一个问题要比直接求解容易。摊还分析将用到这个思路。我们将引入一个附加变量，叫作位势（potential），有了它可以证明以前很难证明的一些结果。

11.2 二项队列

下面考察的第一个数据结构是第6章中的二项队列，现在进行简要的复习。我们知道，二项树 $B_0$ 是一棵单结点树，且对于 $k > 0$ ，二项树 $B_k$ 通过将两棵二项树 $B_{k-1}$ 合并到一起而得到。二项树 $B_0$ 到 $B_4$ 如图11-1所示。

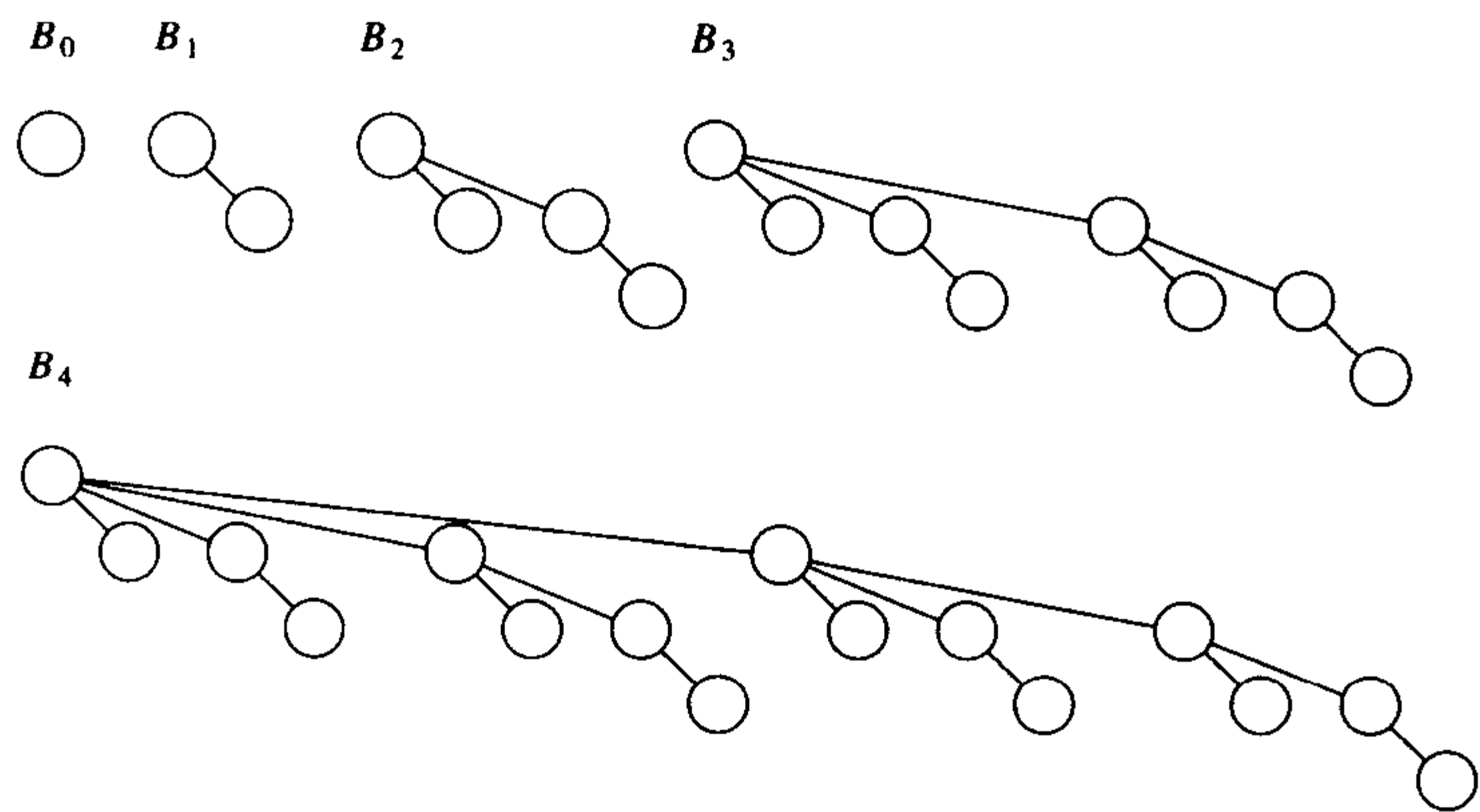


图11-1 二项树 $B_0$ 、 $B_1$ 、 $B_2$ 、 $B_3$ 和 $B_4$

一棵二项树的结点的秩等于其儿子结点的个数；特别地， $B_k$ 的根结点的秩为 $k$ 。二项队列是堆序的二项树的集合，在这个集合中对于任意的 $k$ 最多有一棵二项树 $B_k$ 。图11-2所示为两个二项队列 $H_1$ 和 $H_2$ 。

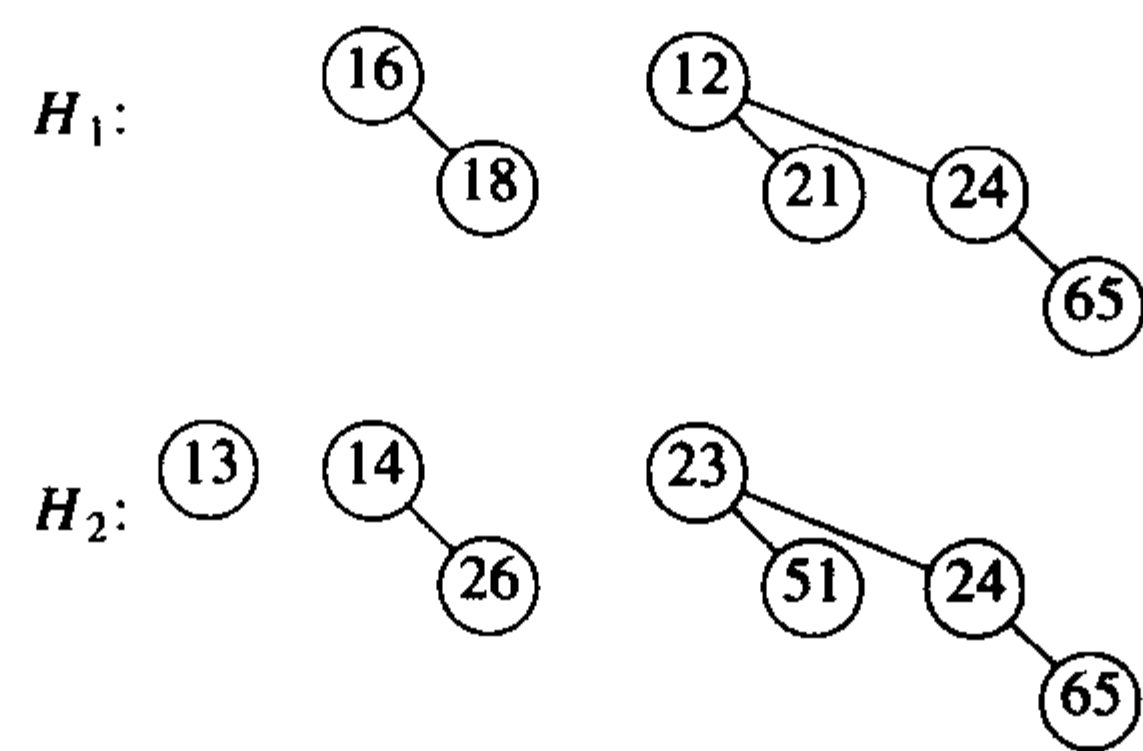
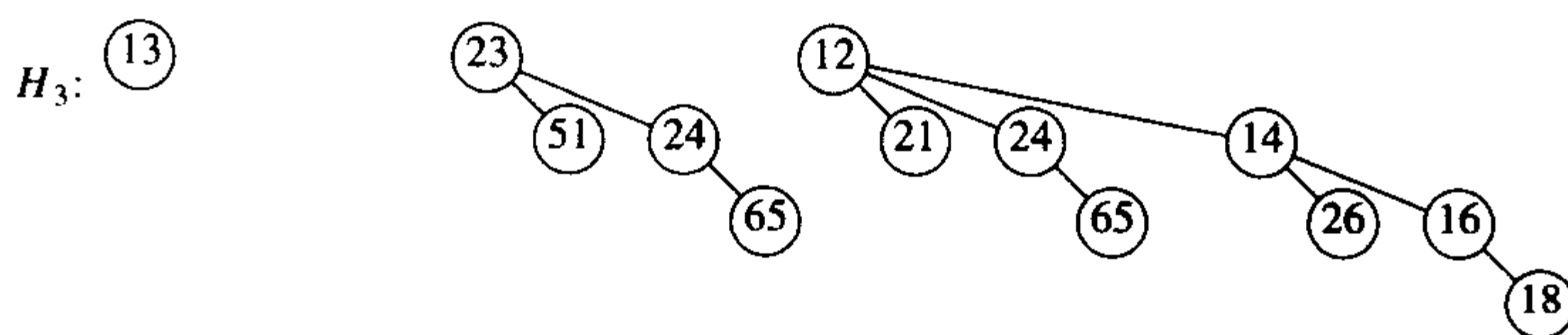


图11-2 两个二项队列 $H_1$ 和 $H_2$

最重要的操作是merge（合并）。为了合并两个二项队列，需要执行类似于二进制整数加法的操作：在任一阶段，可以有零、一、二或可能三棵 $B_k$ 树，它依赖于两个优先队列是否包含一棵 $B_k$ 树以及是否有一棵 $B_k$ 树从前一步转入。如果有零棵或一棵 $B_k$ 树，那么它就作为一棵树放到合并后的二项队列中；如果有两棵 $B_k$ 树，那么它们被合并成一棵 $B_{k+1}$ 树并且并入到结果中；如果有三棵 $B_k$ 树，那么将一棵作为树放入到二项队列中而另两棵树则合并成一棵树且并入到结果中。 $H_1$ 和 $H_2$ 合并的结果如图11-3所示。

图11-3 二项队列 $H_3$ : 合并 $H_1$ 和 $H_2$ 的结果

插入操作通过创建一个单结点二项队列并执行一次merge来完成。做这项工作所用的时间为 $M+1$ , 其中 $M$ 代表不在该二项队列中的二项树 $B_M$ 的最小型号。因此, 向一个有一棵 $B_0$ 树但没有 $B_1$ 树的二项队列进行的插入操作需要两步。删除最小元通过把最小元除去并将原二项队列分裂成两个二项队列, 然后再将它们合并来完成。第6章给出了对这些操作的比较详细的解释。

492  
493

首先考虑一个非常简单的问题。假设想要建立一个含有 $N$ 个元素的二项队列。我们知道, 建立一个含有 $N$ 个元素的二叉堆可以以 $O(N)$ 时间完成, 因此我们希望对二项队列也有一个类似的界。

**断言**  $N$ 个元素的二项队列可以通过 $N$ 次相继插入而以时间 $O(N)$ 建成。

这个断言如果成立, 那么它就给出一个极其简单的算法。因为每次插入的最坏情形时间是 $O(\log N)$ , 所以这个断言是否成立并不是显而易见的。考虑到如果将该算法应用到二叉堆, 则运行时间将是 $O(M \log N)$ 。

要想证明这个断言, 可以直接进行计算。为了测出运行时间, 我们将每次插入的代价定义为一个时间单位加上每一步链接的一个附加单元。将所有插入的时间代价求和就得到总的运行时间。这个总的时间为 $N$ 个单元加上总的链接步数。第一、第三、第五以及所有编号为奇数的步不需要链接步骤, 因为在插入时 $B_0$ 不出现。因此, 有一半的插入不需要链接, 四分之一的插入只需要一次链接(第二、第六、第十次插入等), 八分之一的插入需要两次链接, 依次类推。可以把所有这些加起来并确定用 $N$ 作为链接步数的界, 从而证明该断言。不过, 当试图分析一系列不仅仅是插入的操作时, 这种蛮力计算无助于其后的进一步分析, 因此我们将使用另外一种方法来证明这个结果。

考虑一次插入的结果。如果在插入时不出现 $B_0$ 树, 那么使用与上面相同的计数方法可知这次插入的总代价是一个时间单位。现在, 插入的结果有了一棵 $B_0$ 树, 这样, 我们已经把一棵树添加到二项树的森林中。如果存在一棵 $B_0$ 树, 但是没有 $B_1$ 树, 那么插入花费两个时间单位。新的森林将有一棵 $B_1$ 树但不再有 $B_0$ 树, 因此在森林中树的数目并没有变化。花费三个时间单位的一次插入将创建一棵 $B_2$ 树但消除一棵 $B_0$ 和 $B_1$ 树, 这导致在森林中净减少一棵树。事实上, 容易看到, 一般说来花费 $c$ 个时间单位的一次插入导致在森林中净增加 $2-c$ 棵树, 这是因为创建了一棵 $B_{c-1}$ 树而消除了所有的 $B_i$ 树,  $0 \leq i < c-1$ 。因此, 代价昂贵的插入操作删除一些树, 而低廉的插入却创建一些树。

令 $C_i$ 是第 $i$ 次插入的代价,  $T_i$ 为第 $i$ 次插入之后树的棵数,  $T_0 = 0$ 为树的初始棵数。此时得到不变式

$$C_i + (T_i - T_{i-1}) = 2 \quad (11-1)$$

于是有

$$C_1 + (T_1 - T_0) = 2$$

$$C_2 + (T_2 - T_1) = 2$$

⋮

$$C_{N-1} + (T_{N-1} - T_{N-2}) = 2$$

$$C_N + (T_N - T_{N-1}) = 2$$

把所有这些方程加起来，则大部分的 $T_i$ 项被消去，最后剩下

494

$$\sum_{i=1}^N C_i + T_N - T_0 = 2N$$

或等价地，

$$\sum_{i=1}^N C_i = 2N - (T_N - T_0)$$

考虑到 $T_0 = 0$ 以及 $N$ 次插入后树的棵数 $T_N$ 确实非负，因此 $(T_N - T_0)$ 非负。于是

$$\sum_{i=1}^N C_i \leq 2N$$

这就证明了断言。

在buildBinomialQueue例程运行期间，每一次插入有一个最坏情形运行时间 $O(\log N)$ ，但是由于整个例程最多用到 $2N$ 个时间单位，因此这些插入的行为就像是每次使用不超过两个单元的时间。

这个例子阐明了我们将要使用的一般技巧。数据结构在任一时刻的状态由称为位势的函数给出。这个位势函数不由程序保存，而是一个计数装置，该装置将帮助进行分析。当一些操作花费少于允许使用的时间时，则没有用到的时间就以一个更高位势的形式“存储”起来。在我们的例子中，数据结构的位势就是树的棵数。在上面的分析中，当有一些插入只用到一个单位而不是规定的两个单元的时候，则这个额外的单元通过增加位势而被存储起来以备后用。当操作超出规定的时间时，则超出的时间通过位势的减少来计算。可以把位势看成是一个储蓄账户。如果一次操作的时间少于指定的时间，那么这个差额就被存储起来以备后面更昂贵的操作使用。图11-4所示为buildBinomialQueue对一系列插入操作所使用的累积的运行时间。可以看到，运行时间从不超过 $2N$ 而且在任一次插入后二项队列中的位势计量着存储量。

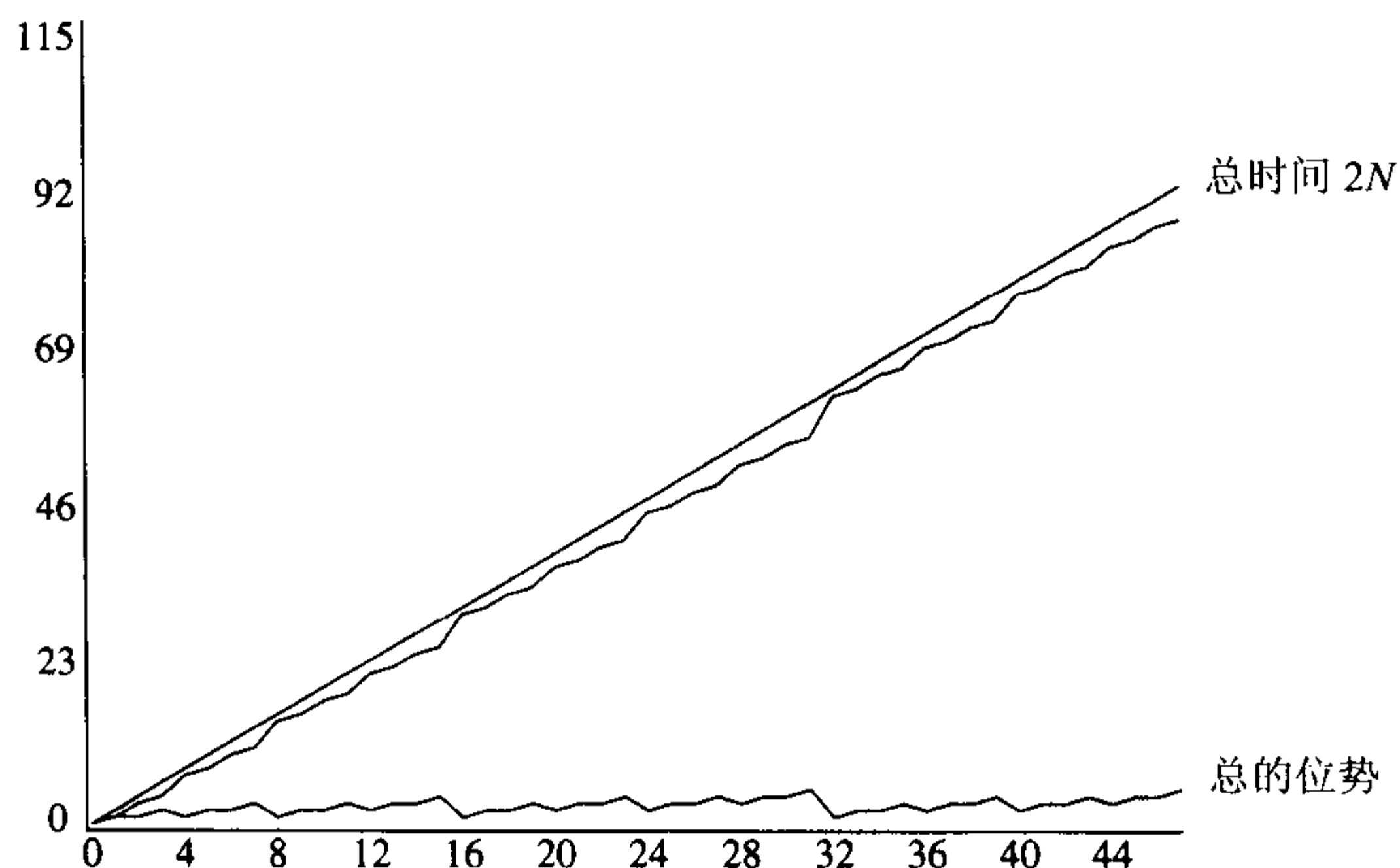


图11-4 连续 $N$ 次insert

一旦选定位势函数，就可写出主要的方程：

$$T_{\text{actual}} + \Delta \text{ 位势} = T_{\text{amortized}} \quad (11-2)$$

$T_{\text{actual}}$ 是一次操作的实际时间，代表执行一次特定操作需要的精确时间量。例如，在二叉查找树中，

执行一次  $\text{find}(x)$  的实际时间是1加上包含  $x$  的结点的深度。如果我们对整个序列把基本方程加起来, 并且最后的位势至少像初始位势一样大, 那么摊还时间就是在操作序列执行期间所用到的实际时间的一个上界。注意, 当  $T_{\text{actual}}$  在从一个操作到另一操作变化时,  $T_{\text{amortized}}$  却是稳定的。

选择位势函数以确保一个有意义的界是一项艰难的工作, 不存在一种实用的方法。一般说来, 在尝试过许多位势函数以后才能够找到一个合适的函数。不过, 上面的讨论提出一些规则, 这些规则告诉我们好的位势函数所具有的一些性质。位势函数应该:

495

- 总假设它的最小值位于操作序列的开始处。选择位势函数的一种常用方法是保证位势函数初始值为0, 而且总是非负的。本章讨论的所有例子都使用这种策略。
- 消去实际时间中的一项。在我们的例子中, 如果实际的开销是  $c$ , 那么位势为  $2-c$ 。把这些加起来就得到摊还开销是2, 这在图11-5中表出。

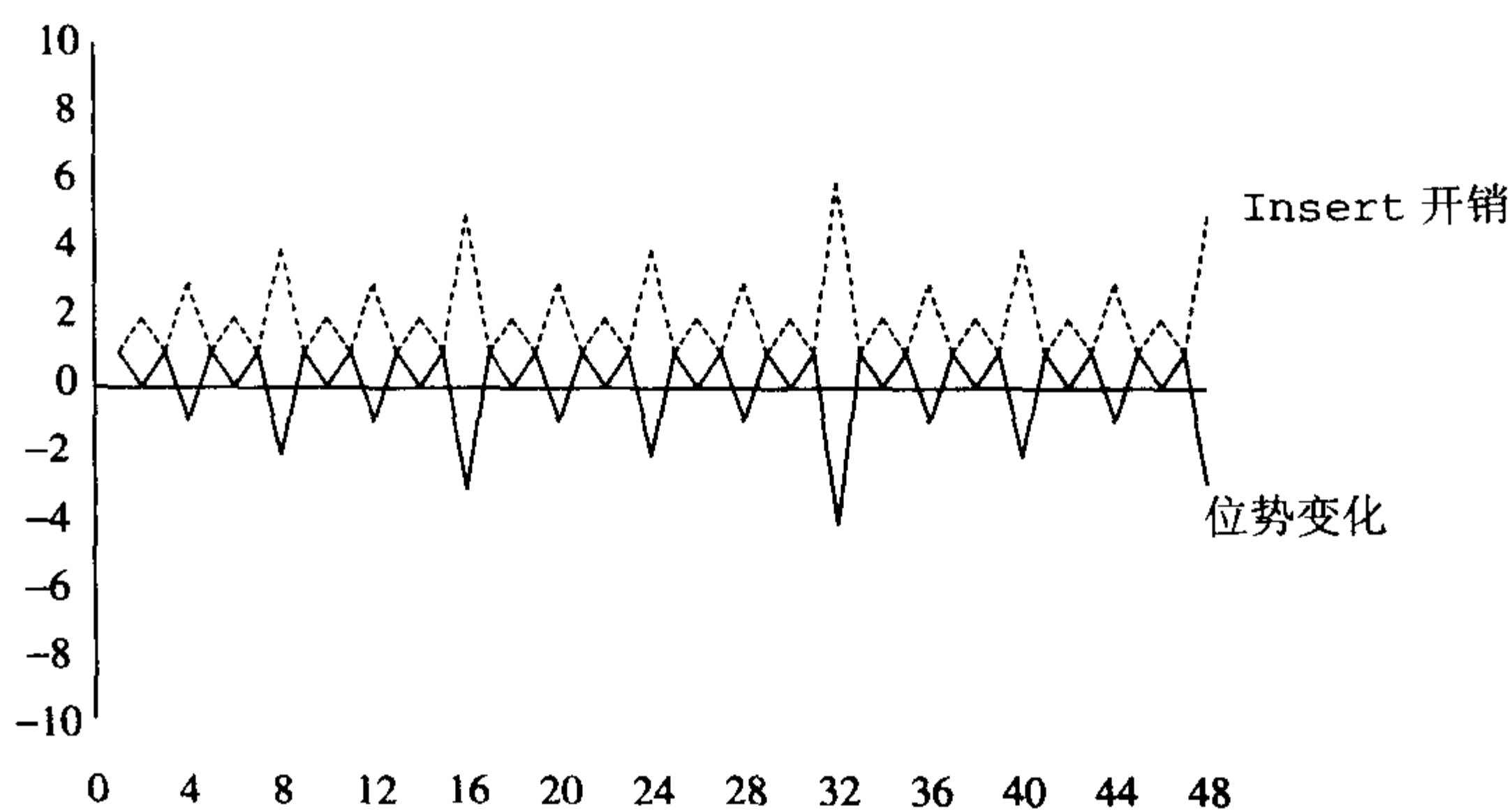


图11-5 在一系列操作中插入的开销和每一次操作的位势变化

下面对二项队列操作进行完整的分析。

**定理11.1**  $\text{insert}$ 、 $\text{deleteMin}$ 以及 $\text{merge}$ 对于二项队列的摊还运行时间分别是  $O(1)$ 、 $O(\log N)$  和  $O(\log N)$ 。

**证明** 位势函数是树的棵数。初始的位势函数为0, 且位势总是非负的, 因此摊还时间是实际时间的一个上界。对  $\text{insert}$  的分析从上面的论证可以得到。对于  $\text{merge}$ , 假设两棵树分别有  $N_1$  和  $N_2$  个结点以及对应的  $T_1$  和  $T_2$  棵树, 令  $N = N_1 + N_2$ , 执行合并的实际时间为  $O(\log(N_1) + \log(N_2)) = O(\log N)$ 。在合并之后, 最多可能存在  $\log N$  棵树, 因此位势最多可以增加  $O(\log N)$ , 这就给出一个摊还的界  $O(\log N)$ 。  $\text{deleteMin}$  操作的界可用类似的方法得到。 ■

496

## 11.3 斜堆

二项队列的分析可以算是一个容易的摊还分析示例。下面考察斜堆。像许多例子一样, 一旦找到正确的位势函数, 分析起来就容易了。困难在于选择一个有意义的位势函数。

对于斜堆, 我们知道关键的操作是合并。为了合并两个斜堆, 把它们的右路径合并并使之成为新的左路径。对于新路径上的每一个结点, 除去最后一个外, 原来的左子树作为右子树而附于其上。在新的左路径上的最后结点已知没有右子树, 因此给它一棵右子树就不明智了。我们所要考虑的界不依赖于这个例外, 而如果例程是递归地编写的, 那么这又是自然要发生的情况。图11-6所示为合并两个斜堆后的结果。



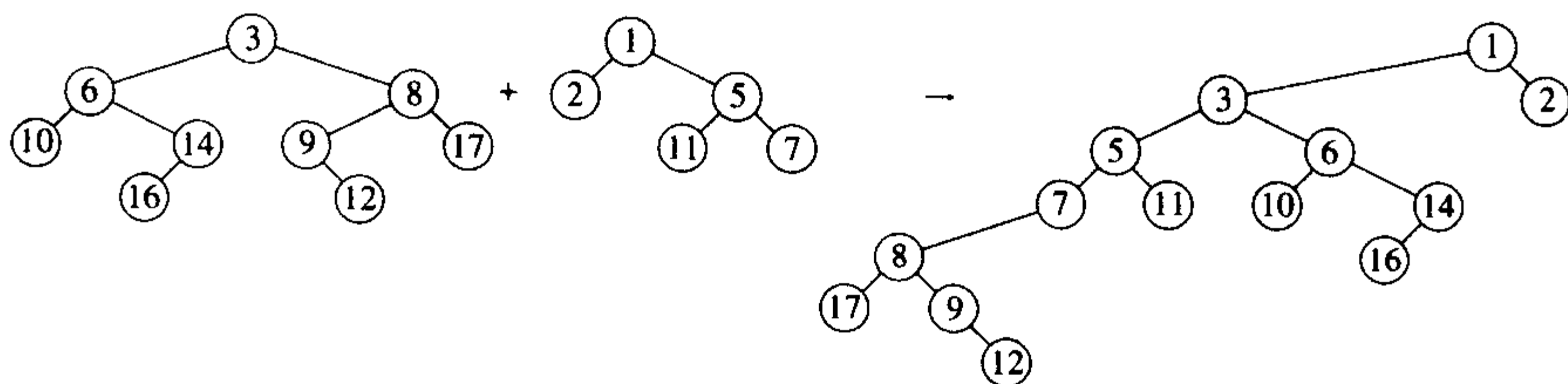


图11-6 合并两个斜堆

497

设有两个斜堆 $H_1$ 和 $H_2$ 并在各自的右路径上分别有 $r_1$ 和 $r_2$ 个结点。此时，执行合并的实际时间与 $r_1 + r_2$ 成正比，因此我们将省去大 $O$ 记法而对右路径上的每一个结点取一个单位的时间。由于这些堆没有固定的结构，因此两个堆的所有结点都位于右路径上的情况是有可能发生的，而这将给出合并两个堆的最坏情形的界 $\Theta(N)$ （练习11.3要求构造一个例子）。我们将证明合并两个斜堆的摊还时间为 $O(\log N)$ 。

需要的是能够获得斜堆操作效果的某种类型的位势函数。已知，一次merge的效果是处在右路径上的每一个结点都被移到左路径上，而其原左儿子变成新的右儿子。一种想法是把每一个结点分类为右结点或左结点，这要看结点是否是右儿子来定，这时把右结点的个数作为位势函数。虽然位势初始为0并且总是非负的，但是问题在于这种位势在一次合并后并不减少从而不能恰当地反映数据结构中的储备量。这样的结果是该位势函数不能够用来证明所要求的界。

一个类似的想法是把结点分成重结点或轻结点，这要看任一结点的右子树上的结点是否比左子树上的结点多来确定。

**定义** 一个结点 $p$ ，如果其右子树的后裔数至少是该 $p$ 的后裔总数的一半，则称结点 $p$ 是重的，否则称之为轻的。注意，一个结点的后裔个数包括该结点本身。

例如，图11-7表示一个斜堆。值为15、3、6、12和7的结点是重结点，而其他的结点都是轻结点。

我们将要使用的位势函数是这些堆（的集合）中的重结点的个数。看起来这可能是一种好的选择，因为一条长的右路径将包含非常多的重结点。由于这条路径上的结点将要交换它们的子结点，因此这些结点将被转变成合并结果中的轻结点。

498

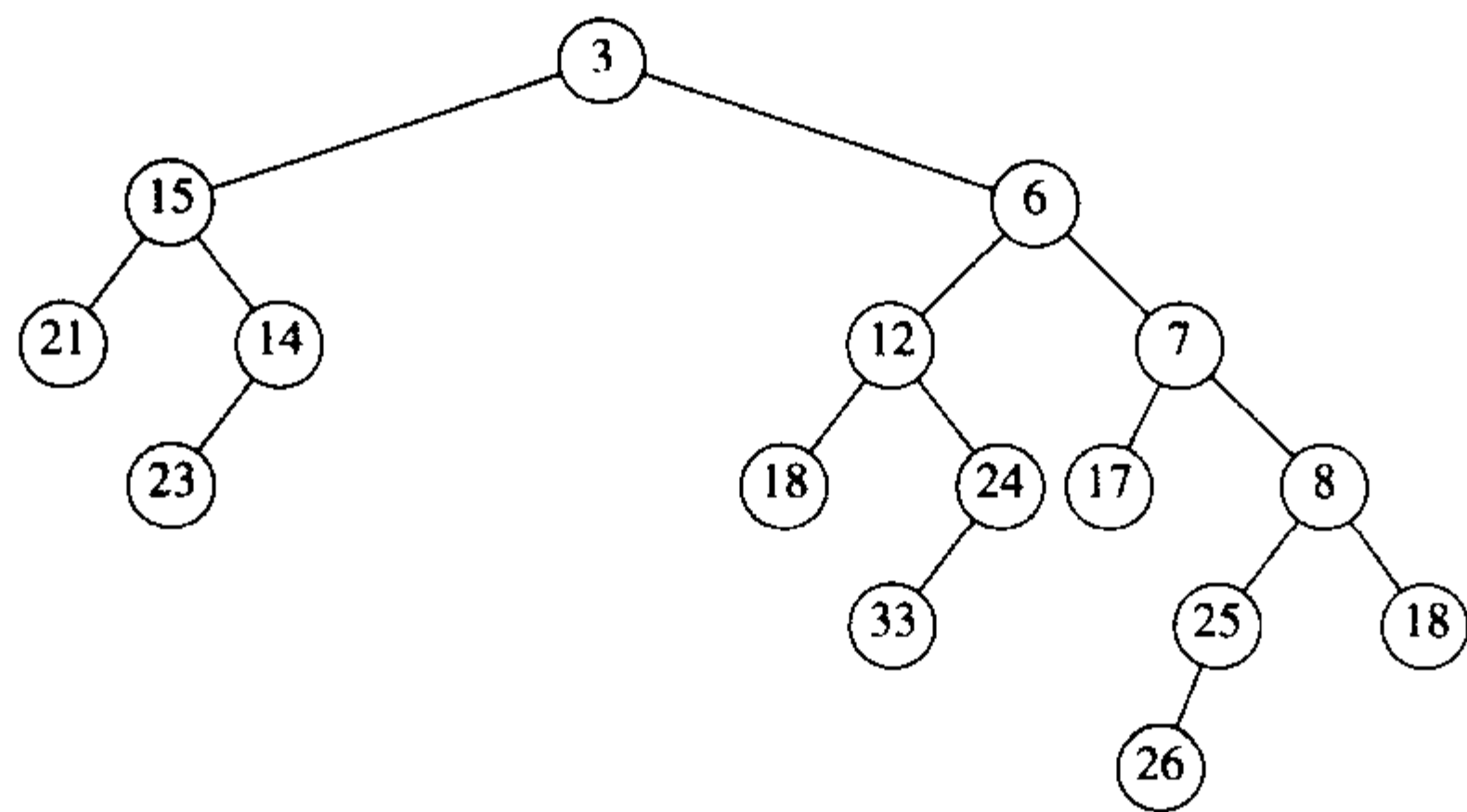


图11-7 斜堆——重结点是3、6、7、12和15

**定理11.2** 合并两个斜堆的摊还时间为 $O(\log N)$ 。

**证明** 令 $H_1$ 和 $H_2$ 为两个堆，分别具有 $N_1$ 和 $N_2$ 个结点。设 $H_1$ 的右路径有 $l_1$ 个轻结点和 $h_1$ 个重结点，共有 $l_1 + h_1$ 个结点。同样， $H_2$ 在其右路径上有 $l_2$ 个轻结点和 $h_2$ 个重结点，共有 $l_2 + h_2$ 个结点。

如果我们采用约定：合并两个斜堆的开销是它们右路径上结点的总数，那么执行合并的实际时间就是 $l_1 + l_2 + h_1 + h_2$ 。现在，重/轻状态可能改变的结点只是那些最初位于右路径上的结点（并最后出现在左路径上），因为再没有别的结点的子树被交换。这可见于图11-8中的例子。

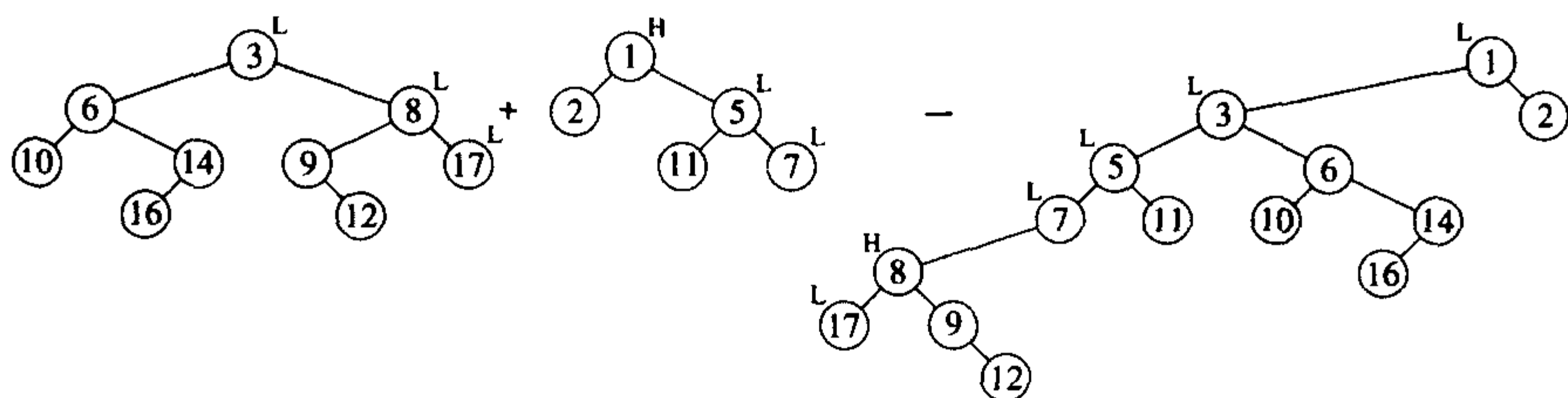


图11-8 合并后重/轻状态的变化

如果一个重结点最初是在右路径上，那么合并后它必然成为一个轻结点。位于右路径上的其余结点是轻结点，它们可能变成也可能不变成重结点，但是由于我们要证明一个上界，因此必须假设最坏的情形，即它们都变成了重结点并使得位势增加。此时，重结点个数的净变化最多为 $l_1 + l_2 - h_1 - h_2$ 。把实际时间和位势的变化（式（11-2））加起来则得到一个摊还界 $2(l_1 + l_2)$ 。

现在必须证明 $l_1 + l_2 = O(\log N)$ 。由于 $l_1$ 和 $l_2$ 是原右路径上轻结点的个数，而一个轻结点的右子树小于以该轻结点为根的树的大小的一半，由此直接推出右路径上轻结点的个数最多为 $\log N_1 + \log N_2$ ，这就是 $O(\log N)$ 。

注意到初始的位势为0而且位势总是非负的，所以证明也就完成了。验证这一点很重要，因为否则摊还时间就不能成为实际的时间界而且也就没有意义了。■

由于insert和deleteMin操作基本上就是一些merge，因此它们的摊还界也是 $O(\log N)$ 。

## 11.4 斐波那契堆

9.3.2节介绍了如何使用优先队列来改进Dijkstra最短路径算法粗略的运行时间 $O(|V|^2)$ 。重要的观察结论是运行时间被 $|E|$ 次decreaseKey操作和 $|V|$ 次insert和deleteMin操作所控制。这些操作发生在大小最多为 $|V|$ 的集合上。通过使用二叉堆，所有这些操作花费 $O(\log |V|)$ 时间，因此Dijkstra算法最后的界可以减到 $O(|E| \log |V|)$ 。

499

为了降低这个时间界，必须改进执行decreaseKey操作所需要的时间。6.5节所描述的 $d$ 堆给出对于decreaseKey操作以及insert的 $O(\log_d |V|)$ 时间界，但对deleteMin的界却是 $O(d \log_d |V|)$ 。通过选择 $d$ 来平衡带有 $|V|$ 次deleteMin操作的 $|E|$ 次decreaseKey操作的开销，并考虑到 $d$ 必须总是至少为2，可以看到 $d$ 的一个好的选择是

$$d = \max(2, \lfloor |E|/|V| \rfloor)$$

它把Dijkstra算法的时间界改进到

$$O(|E| \log_{(2 + \lfloor |E|/|V| \rfloor)} |V|)$$

斐波那契堆（Fibonacci heap）是以 $O(1)$ 摊还时间支持所有基本的堆操作的一种数据结构，但deleteMin和delete除外，它们花费 $O(\log N)$ 的摊还时间。立即得出，在Dijkstra算法中的那些堆操作将总共需要 $O(|E| + |V| \log |V|)$ 的时间。

斐波那契堆<sup>1</sup>通过添加下面两个新观念推广了二项队列：

- decreaseKey的一种不同的实现方法：我们以前看到的方法是把元素朝向根结点上滤。对于这种方法似乎没有理由期望 $O(1)$ 的摊还时间界，因此需要一种新的方法。
- 懒惰合并 (lazy merging)：只有两个堆需要合并时才进行合并，这类似于懒惰删除。对于懒惰合并，merge是低廉的，但是因为懒惰合并实际上并不把树结合在一起，所以deleteMin操作可能会遇到许多的树，从而使这种操作的代价高昂。任何一次deleteMin都可能花费线性时间，但是它总能够把时间归咎到前面的一些merge操作中去。特别地，一次昂贵的deleteMin必须在其前面有大量的非常低廉的merge操作，它们能够存储额外的位势。

11.4.1 切除左式堆中的结点

在二叉堆中，decreaseKey操作是通过降低结点的值然后将其朝着根上滤直到建成堆序来实现的。在最坏情形下，它花费 $O(\log N)$ 时间，这是平衡树中通向根的最长路径的长。

如果代表优先队列的树不具有 $O(\log N)$ 的深度，那么这种方法就不适用。例如，若将这种方法用于左式堆，则decreaseKey操作可能花费 $\Theta(N)$ 时间，如图11-9中的例子所示。

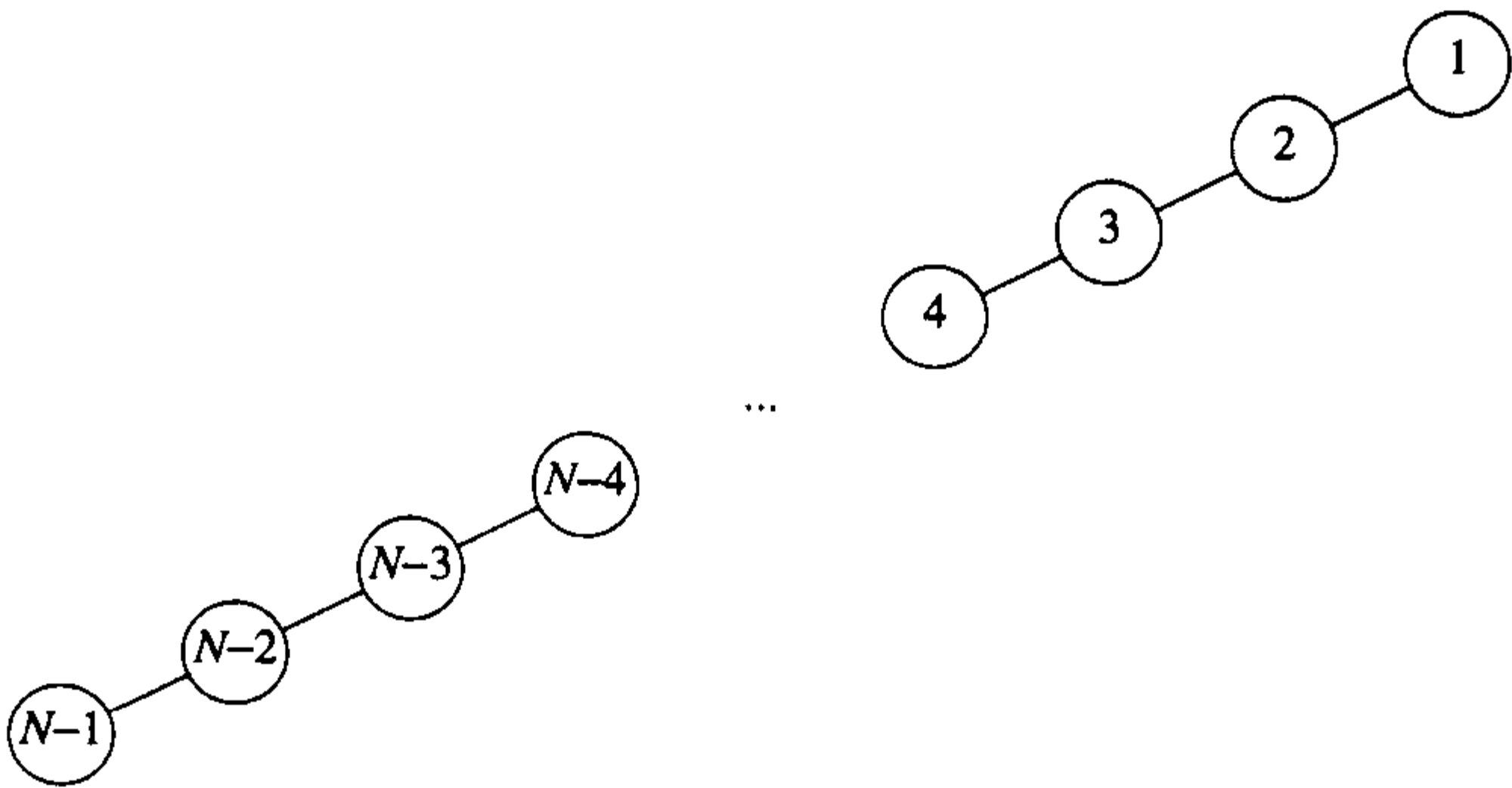


图11-9 通过上滤将 $N-1$ 递减到0花费 $\Theta(N)$ 时间

500

可以看到，对于左式堆来说，decreaseKey操作需要其他的策略。见图11-10中左式堆的例子。假设想要将值为9的键减到0。若对该堆做变动，则必将引起堆序的破坏，这种破坏在图11-11中用虚线标示。

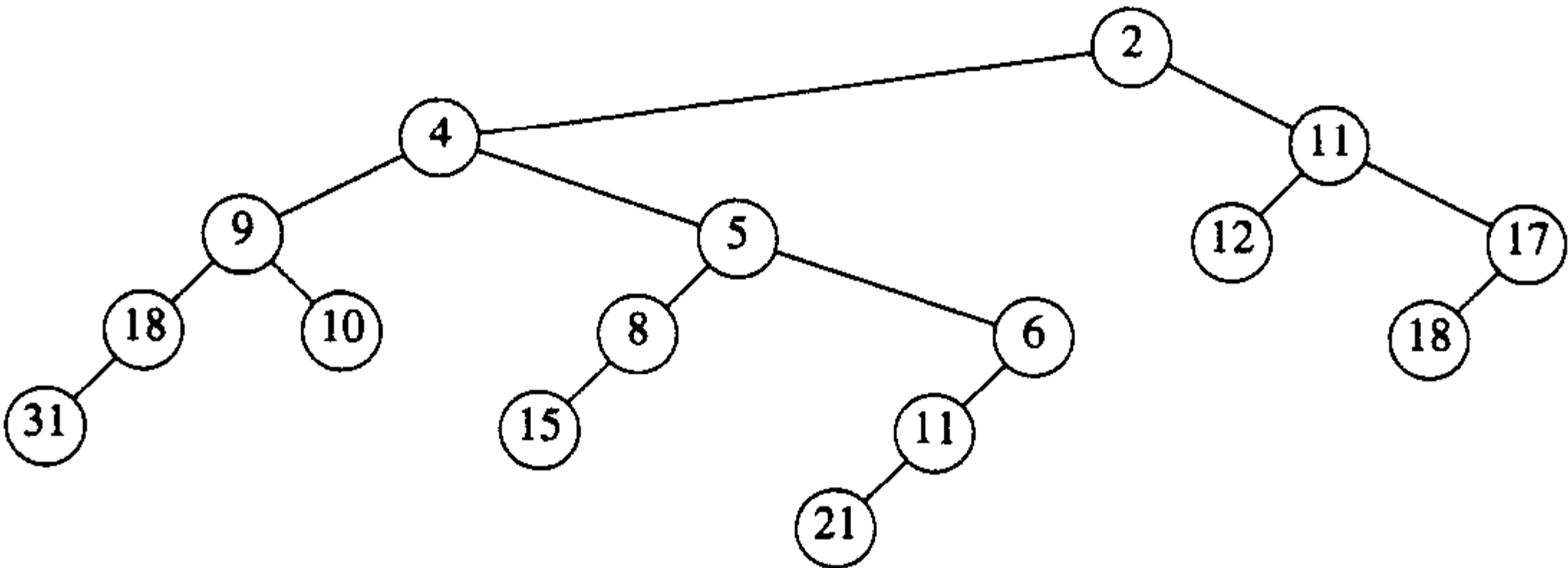


图11-10 样本左式堆H

1. 这个名字来自于这种数据结构的一个性质，本节的后面将要证明它。

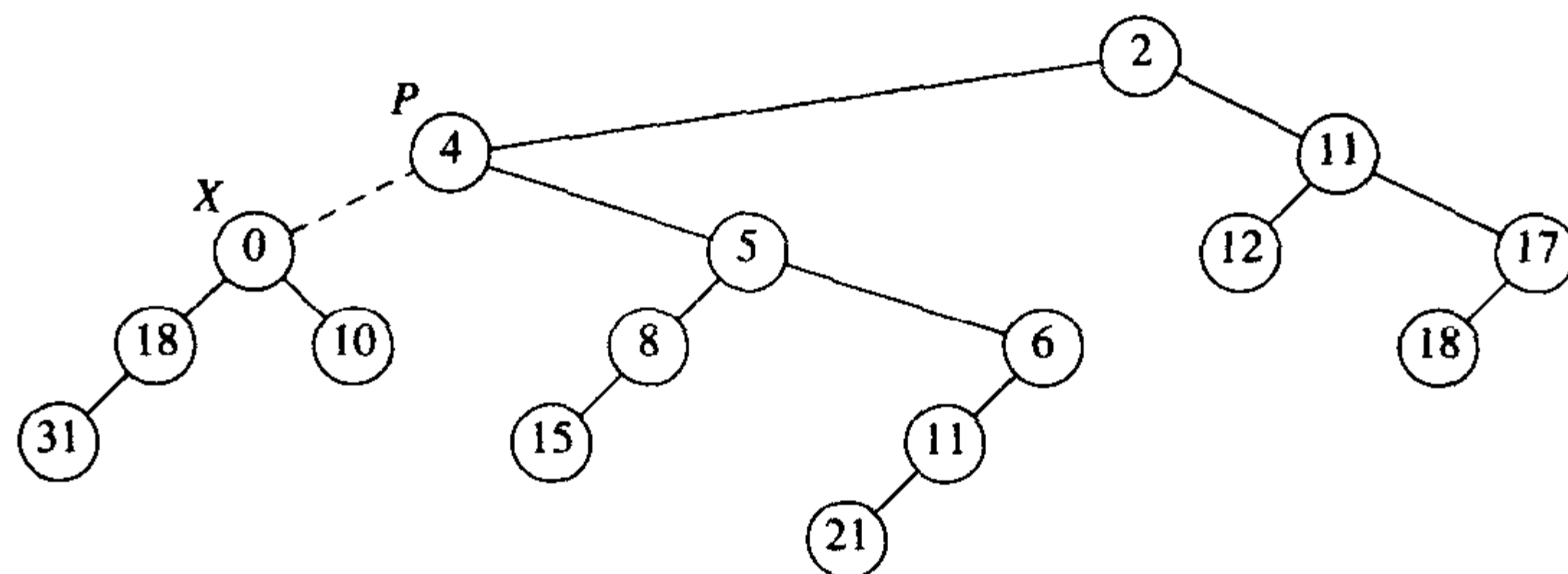


图11-11 将9降到0引起堆序的破坏

我们不想把0上滤到根，因为正如前面已经介绍的，存在一些情况使得这样做代价太大。解决的办法是将堆沿着虚线切开，于是得到两棵树，然后再把这两棵树合并成一棵。令 $X$ 为要执行decreaseKey操作的结点，令 $P$ 为它的父结点。在切断以后得到两棵树，即根为 $X$ 的 $H_1$ 和 $T_2$ ， $T_2$ 是原来的树除去 $H_1$ 后得到的树。具体情况如图11-12所示。

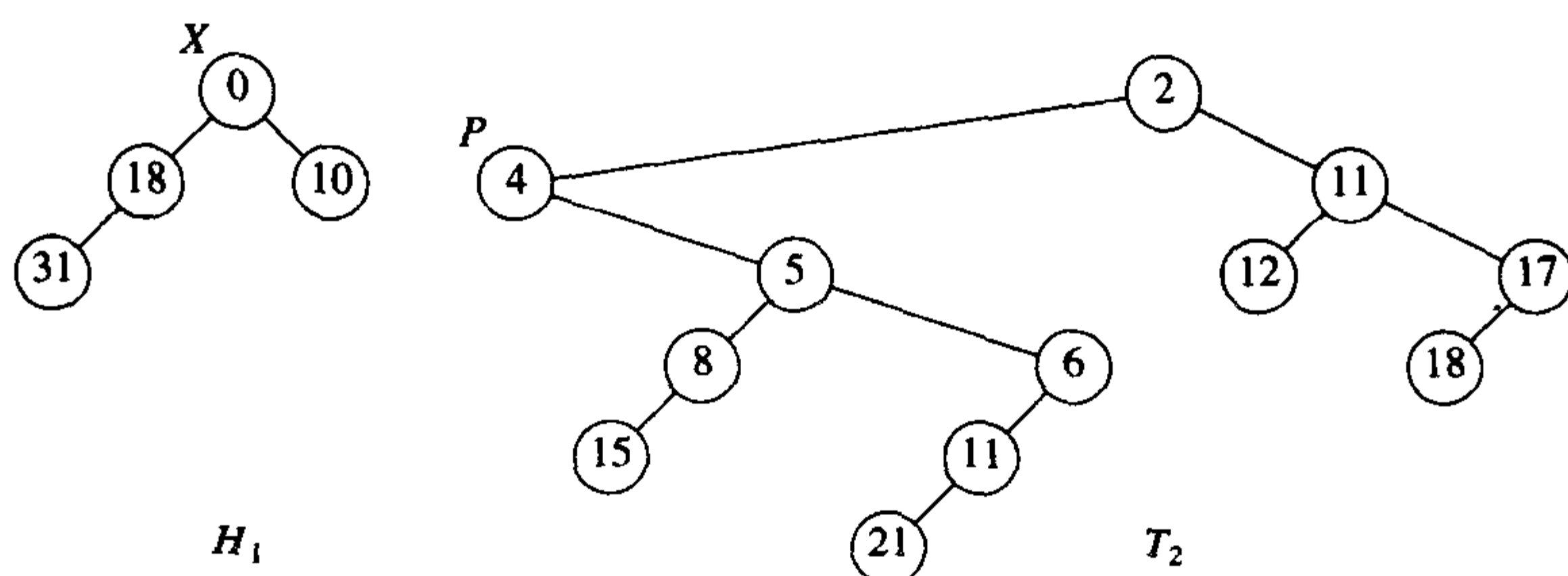


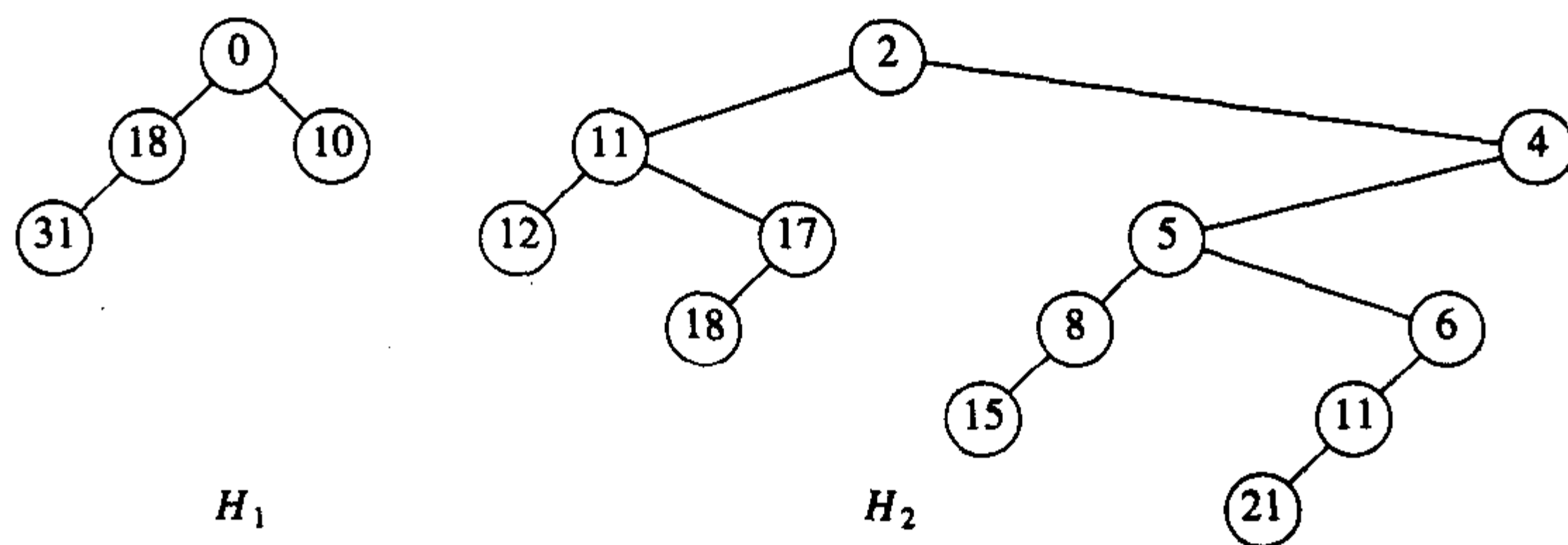
图11-12 切断之后得到的两棵树

如果这两棵树都是左式堆，那么它们可以以时间 $O(\log N)$ 合并，整个操作也就完成了。容易看出， $H_1$ 是左式堆，因为没有结点的后裔发生变化。由于它的所有结点原本就满足左式堆的性质，因此现在必然满足。

501

然而，这种方案似乎还是行不通，因为 $T_2$ 未必是左式堆。不过，容易恢复左式堆的性质，这要用到下列两个观察到的结论：

- 只有从 $P$ 到 $T_2$ 的根的路径上的结点可能破坏左式堆的性质；它们可以通过交换子结点来调整。
- 由于最大右路径长最多有 $\lfloor \log(N+1) \rfloor$ 个结点，因此只需检查从 $P$ 到 $T_2$ 的根的路径上的前 $\lfloor \log(N+1) \rfloor$ 个结点。图11-13显示了 $H_1$ 和将 $T_2$ 转变成左式堆后的 $H_2$ 。

图11-13 将 $T_2$ 转变成左式堆 $H_2$ 后的情形



因为我们能够以 $O(\log N)$ 步将 $T_2$ 转变成左式堆 $H_2$ ，然后合并 $H_1$ 和 $H_2$ ，所以得到一个在左式堆中执行 $\text{decreaseKey}$ 的 $O(\log N)$ 算法。图11-14显示的堆是该例的最后结果。

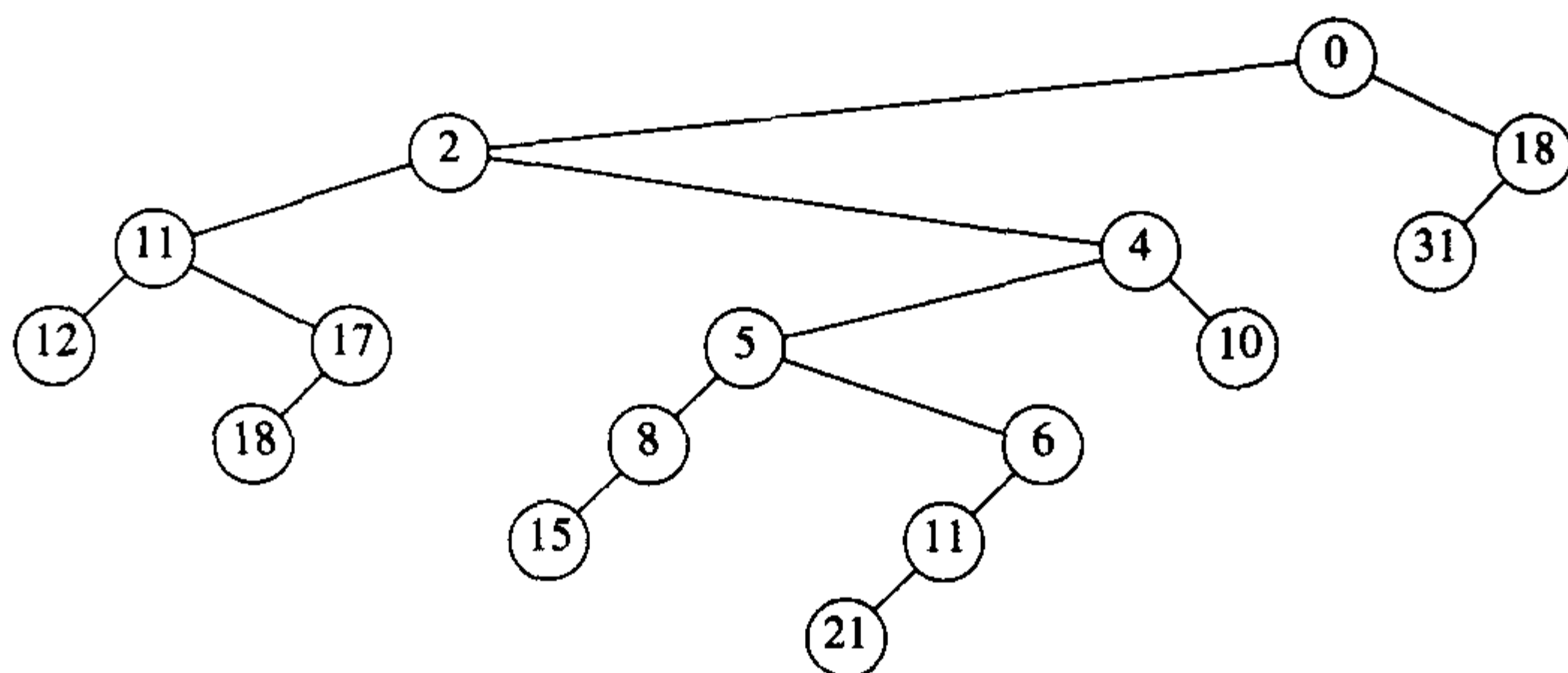


图11-14 通过合并 $H_1$ 和 $H_2$ 来完成操作 $\text{decreaseKey}(X, 9)$

### 11.4.2 二项队列的懒惰合并

**502** 斐波那契堆所使用的第二个想法是懒惰合并 (lazy merging)。我们将把这个想法用于二项队列并证明执行一次merge操作 (还有插入操作，它是一种特殊情形) 的摊还时间为 $O(1)$ 。对于 $\text{deleteMin}$ ，其摊还时间仍然是 $O(\log N)$ 。

这个想法如下：为了合并两个二项队列，只要把两个二项树的列表连在一起，结果得到一个新的二项队列即可。这个新的队列可能含有相同大小的多棵树，因此破坏了二项队列的性质。为了保持一致性，我们把它叫作懒惰二项队列 (lazy binomial queue)。这是一种快速操作，该操作总是花费常数 (最坏情形) 时间。和前面一样，一次插入通过创建一个单结点二项队列并将其合并而完成。区别在于merge是懒惰的。

$\text{deleteMin}$ 操作要麻烦得多，因为此处需要最终把懒惰二项队列转变回到标准的二项队列，不过，正如我们将要证明的，它仍然花费 $O(\log N)$ 的摊还时间——但不像以前是 $O(\log N)$ 最坏情形时间。为了执行一次 $\text{deleteMin}$ ，我们找出 (并最终返回) 最小元素。如前所述，将它从队列中删除，使得它的每一个儿子都成为一棵新的树。此时通过合并两棵相等大小的树直至不再可能合并为止而把所有的树合并成一个二项队列。

**503**

作为一个例子，图11-15表示一个懒惰二项队列。在懒惰二项队列中，可能有多棵树有相同的大小。为了执行 $\text{deleteMin}$ ，像以前那样删除最小元素，并得到图11-16中的树。

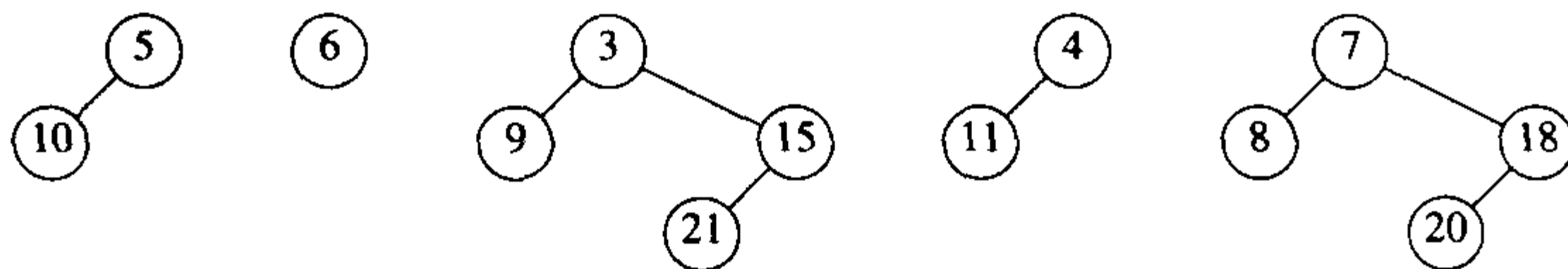


图11-15 懒惰二项队列

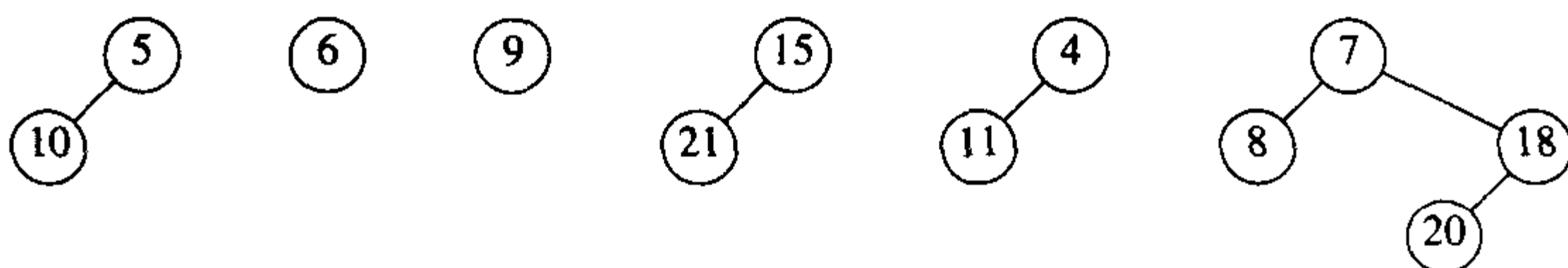


图11-16 删除最小元素 (3) 后的懒惰二项队列

现在我们必须将所有的树合并来得到一个标准的二项队列。标准的二项队列每个秩上最多有一棵树。为了有效地进行这项工作，必须能够以正比于出现的树的棵数 ( $T$ ) 的时间 (或 $\log N$ ,

哪个大用哪个) 完成merge。为此, 构造表的一个数组:  $L_0, L_1, \dots, L_{R_{\max}+1}$ , 其中 $R_{\max}$ 是最大的树的秩。每个表 $L_R$ 包含秩为 $R$ 的所有的树。然后应用图11-17中的过程。

```

1 for( R = 0; R <= ⌊logN⌋; R++ )
2   while( |LR| >= 2 )
3   {
4     Remove two trees from LR;
5     Merge the two trees into a new tree;
6     Add the new tree to LR+1;
7   }

```

图11-17 恢复二项队列的过程

每通过一次过程中的从第4行到第6行的循环, 树的总棵数都要减少1。这意味着, 这部分每次执行都花费常数时间的代码只能够执行 $T-1$ 次, 其中 $T$ 是树的棵数。这里的for循环计数器和while循环末尾的检测花费 $O(\log N)$ 时间, 这使得运行时间成为所要求的 $O(T+\log N)$ 。图11-18显示了该算法对前面二项树的集合的执行情况。

504

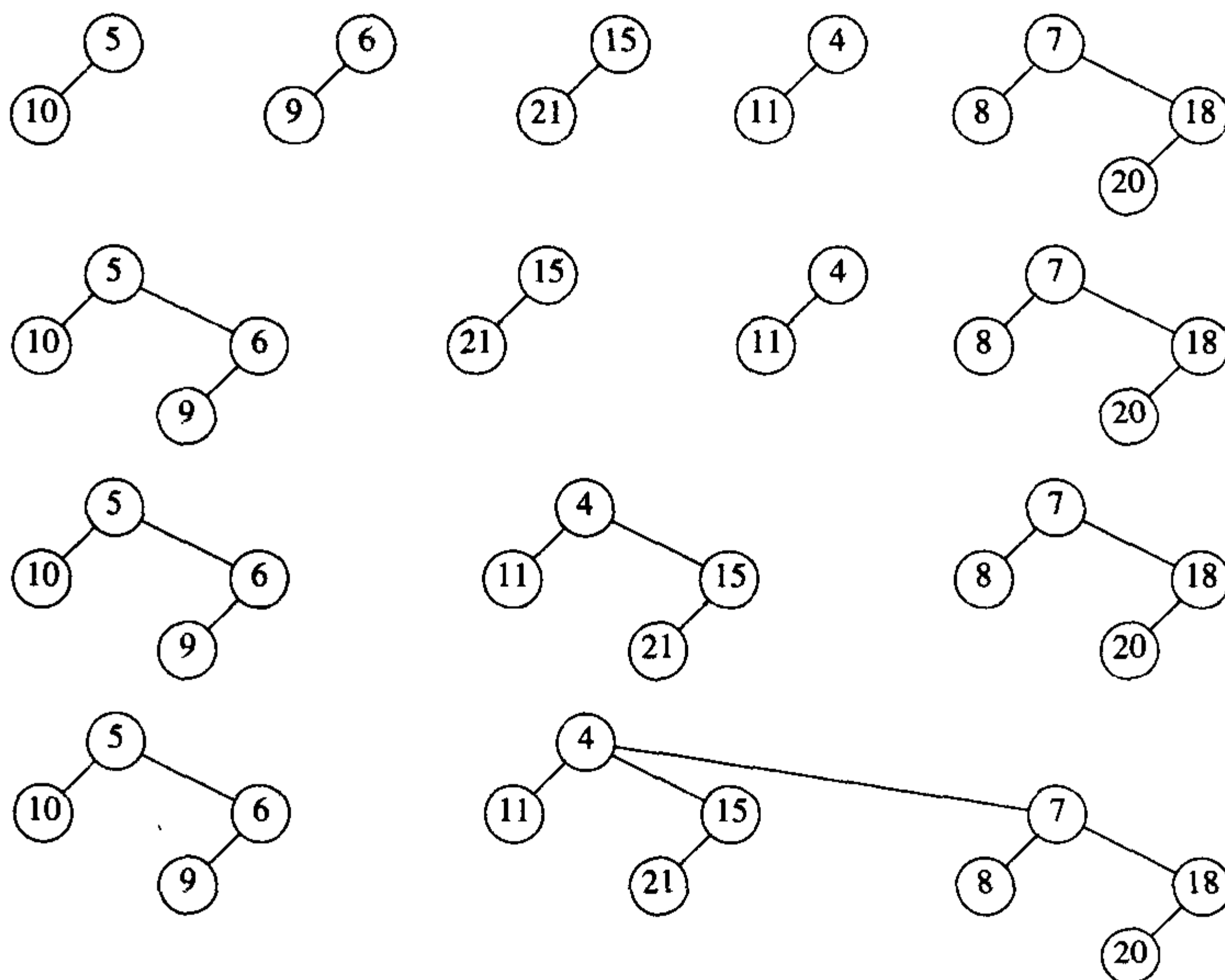


图11-18 把一些二项树合并成一个二项队列

### 懒惰二项队列的摊还分析

为了进行懒惰二项队列的摊还分析, 我们将用到与标准二项队列所使用的相同的位势函数。因此, 懒惰二项队列的位势是树的棵数。

**定理11.3** merge和insert的摊还运行时间对于懒惰二项队列均为 $O(1)$ ; deleteMin的摊还运行时间为 $O(\log N)$ 。

**证明** 这里的位势函数为二项队列集合中树的棵数。初始的位势为0, 而且位势总是非负的。因此, 经过一系列的操作之后, 总的摊还时间是总的实际时间的一个上界。

对于merge操作, 实际时间为常数, 而二项队列的集合中的树的棵数是不变的, 因此, 由式(11.2)可知摊还时间为 $O(1)$ 。

对于insert操作, 实际时间是常数, 而树的棵数最多增加1, 因此摊还时间为 $O(1)$ 。

505

操作deleteMin比较复杂。令 $R$ 为包含最小元素的树的秩，而令 $T$ 是树的棵数。于是，在deleteMin操作开始时的位势为 $T$ 。为执行deleteMin，最小结点的各子结点被分离而成为一棵一棵的树，这就产生了 $T+R$ 棵树，这些树必须要合并成一个标准的二项队列。如果忽略大O记法中的常数<sup>1</sup>，那么根据上面的论述可知，执行该操作的实际时间为 $T + R + \log N$ 。另一方面，一旦做完这些，最多剩下 $\log N$ 棵树，因此位势函数最多增加 $\log N - T$ 。把实际时间和位势的变化加起来得到摊还时间界为 $2\log N + R$ 。由于所有的树都是二项树，因此 $R \leq \log N$ 。这样，得到deleteMin操作的摊还时间界 $O(\log N)$ 。■

### 11.4.3 斐波那契堆操作

正如前面提到的，斐波那契堆将左式堆decreaseKey操作与懒惰二项队列merge操作结合起来。不过，我们不能一点修改也不做就使用这两种操作。问题在于，如果在这些二项树中进行任意切割，那么得到的森林将不再是二项树的集合。因此，每一棵树的秩最多为 $\lfloor \log N \rfloor$ 的结论将不再成立。由于在懒惰二项队列中deleteMin的摊还时间界已被证明是 $2\log N + R$ ，因此，为使deleteMin的界成立需要 $R = O(\log N)$ 成立。

为了保证 $R = O(\log N)$ ，对所有的非根结点应用下述规则：

- 将第一次（因为切除而）失去一个儿子的（非根）结点做上标记。
- 如果被标记的结点又失去另外一个儿子结点，那么将其从它的父结点切除。这个结点现在变成了一棵分离的树的根并且不再被标记。这叫作一次级联切除（cascading cut），因为在一次decreaseKey操作中可能出现多次这种切除。

图11-19显示了在decreaseKey操作之前斐波那契堆中的一棵树。当键为39的结点变成12的时候，堆序被破坏。因此，该结点被从它的父结点中切除，变成了一棵新树的根。由于包含33的结点做了标记，这是它的第二个失去的儿子，从而它也被从其父结点（10）中切除。现在，10也失去了它的第二个儿子，于是它又被从5中切除。这个过程到这里结束，因为5是未做标记的。现在把结点5做上标记，结果如图11-20所示。

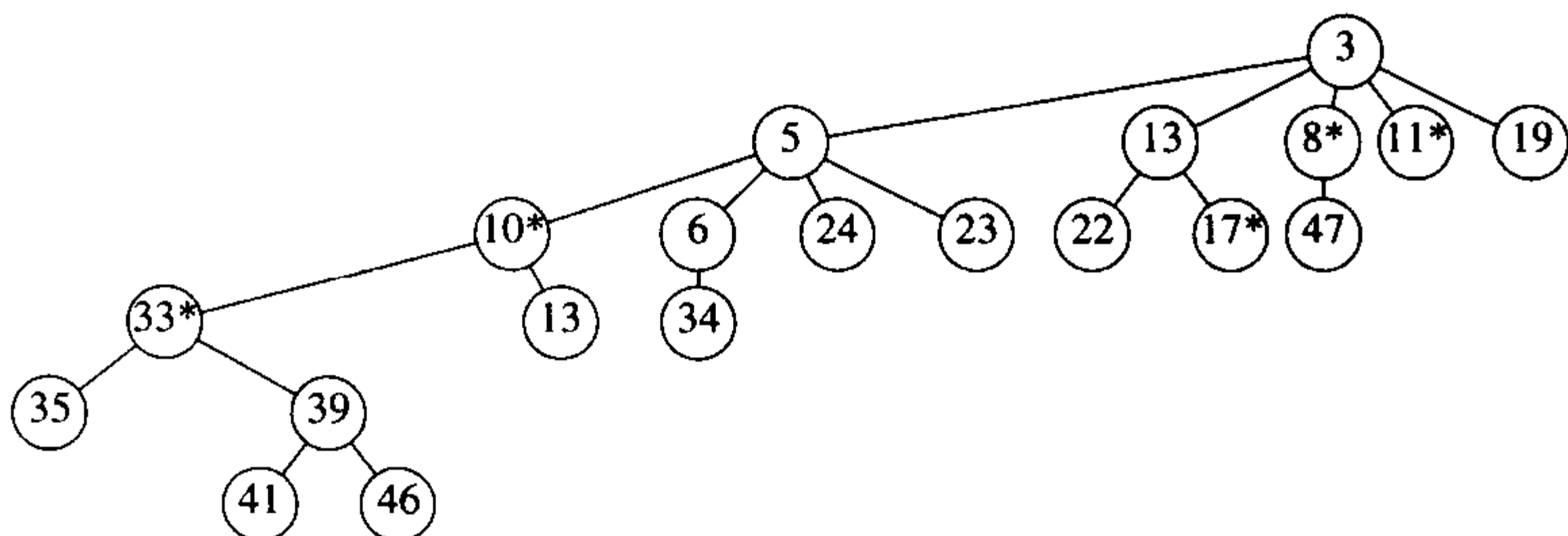


图11-19 将39减成12之前斐波那契堆中的一棵树

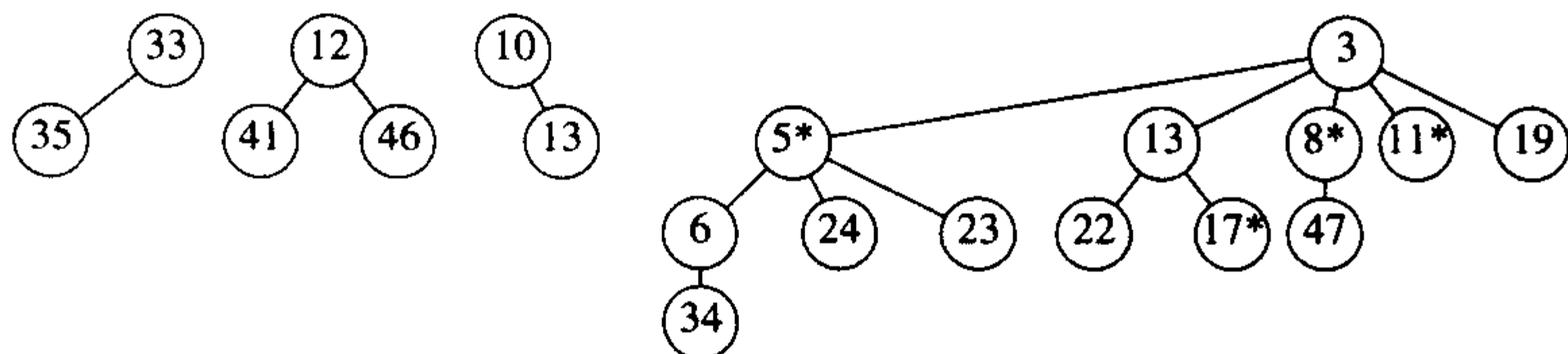


图11-20 在decreaseKey操作之后斐波那契堆的结果

1. 可以这么做，因为我们可以把大O记法中暗含的常数放在位势函数中，并仍可消去这些项，这在证明中是需要的。

注意，过去做过标记的结点10和33不再被标记，因为现在它们都是根结点。这在时间界的证明中是极其重要的。

#### 11.4.4 时间界的证明

注意，标记结点的原因是需要给任一结点的秩 $R$ （儿子的个数）确定一个界。下面证明具有 $N$ 个后裔的任意结点的秩为 $O(\log N)$ 。

506

**引理11.1** 令 $X$ 是斐波那契堆中的任一结点， $c_i$ 为 $X$ 的第 $i$ 个最年轻的儿子，则 $c_i$ 的秩至少是 $i-2$ 。

**证明** 在 $c_i$ 被链接到 $X$ 上的时候， $X$ 已经有（年长的）儿子 $c_1, c_2, \dots, c_{i-1}$ 。于是，当链接到 $c_i$ 时 $X$ 至少有 $i-1$ 个儿子。由于只有当结点有相同的秩的时候它们才链接，由此可知在 $c_i$ 被链接到 $X$ 上的时候 $c_i$ 至少也有 $i-1$ 个儿子。从这时起，它可能至多失去一个儿子，否则它就已经被从 $X$ 切除。因此， $c_i$ 至少有 $i-2$ 个儿子。 ■

从引理11.1容易证明，秩为 $R$ 的任意结点必然有许多后裔。

**引理11.2** 令 $F_k$ 是由 $F_0=1$ 、 $F_1=1$ 以及 $F_k=F_{k-1}+F_{k-2}$ 定义（见1.2节）的斐波那契数。秩为 $R \geq 1$ 的任意结点至少有 $F_{R+1}$ 个后裔（包括它自身）。

**证明** 令 $S_R$ 是秩为 $R$ 的最小的树。显然， $S_0=1$ 和 $S_1=2$ 。根据引理11.1，一棵秩为 $R$ 的树必然含有秩至少为 $R-2, R-3, \dots, 1$ 和 $0$ 的子树，再加上另一棵至少有一个结点的子树。连同 $S_R$ 的根本身一起，这就给出 $S_R = 2 + \sum_{i=0}^{R-2} S_i$  ( $S_R > 1$ ) 的一个最小值。容易证明， $S_R = F_{R+1}$ （练习1.11a）。 ■

507

因为众所周知斐波那契数是以指数增长，所以直接推出具有 $s$ 个后裔的任意结点的秩最多为 $O(\log s)$ 。于是，我们有下面的引理。

**引理11.3** 斐波那契堆中任意结点的秩为 $O(\log N)$ 。

**证明** 直接从上面的讨论得出。 ■

假如我们所关心的只是merge、insert以及deleteMin等操作的时间界，那么现在就可以停止并证明所要的摊还时间界了。当然，斐波那契堆的全部意义在于还要得到一个对于decreaseKey的 $O(1)$ 时间界。

对于一次decreaseKey操作所需要的实际时间是1加上在该操作期间所执行的级联切除的次数。由于级联切除的次数可能会比 $O(1)$ 多很多，为此需要用位势的损失来作为补偿。从图11-20可以看到，树的棵数实际上是随着每次级联切除而增加的，因此必须增强位势函数，使它包含某种在级联切除期间能够递减的成分。注意，不能从位势函数中抛开树的棵数，因为这样就不能够证明merge操作的时间界了。再次观察图11-20可以看到，级联切除引起被标记的结点的个数的减少，因为每个被级联切除分出的结点都变成了未标记的根。由于每个级联切除均花费1个单元的实际时间并将树的位势增加1，因此将每个标记的结点算作2个位势单位。利用这种方法，可以获得一个消除级联切除次数的机会。

**定理11.4** 斐波那契堆对于insert、merge和decreaseKey的摊还时间界均为 $O(1)$ ，而对于deleteMin则是 $O(\log N)$ 。

**证明** 位势是斐波那契堆的集合中树的棵数加上两倍的标记结点数。像通常一样，初始的位



势为0并且总是非负的。于是，经过一系列操作之后，总的摊还时间则是总的实际时间的一个上界。

对于merge操作，实际时间为常数，而树和标记结点的数目是不变的，因此根据式(11.2)，摊还时间为 $O(1)$ 。

对于insert操作，实际时间是常数，树的棵数增加1，而标记结点的个数不变。因此，位势最多增加1，所以摊还时间也是 $O(1)$ 。

508 对于deleteMin操作，令 $R$ 为包含最小元素的树的秩，并令 $T$ 是操作前树的棵数。为执行一次deleteMin，我们再一次将树的儿子分离，得到另外 $R$ 棵新的树。注意，虽然这可以除去一些标记的结点（通过使它们成为未标记的根），但却不能创建另外的标记结点。这 $R$ 棵新树，和其余 $T$ 棵树一起，现在必须合并，根据引理11.3其开销为 $T + R + \log N = T + O(\log N)$ 。由于最多有 $O(\log N)$ 棵树，而标记结点的个数又不可能增加，因此位势的变化最多是 $O(\log N) - T$ 。将实际时间和位势的变化加起来则得到deleteMin的 $O(\log N)$ 摊还时间界。

最后考虑decreaseKey操作。令 $C$ 为级联切除的次数。decreaseKey的实际花费为 $C + 1$ ，它是所执行的切除的总数。第一次（非级联）切除创建一棵新树从而使位势增1。每次级联切除都建立一棵新树，但却把一个标记结点转变成未标记的（根）结点，合计每次级联切除有一个单位的净损失。最后一次切除也把一个未标记结点（在图11-20中这个结点为5）转变成标记结点，这就使得位势增加2。因此，位势总的变化最多是 $3 - C$ 。把实际时间和位势变化加起来则得到总和为4，即 $O(1)$ 。 ■

## 11.5 伸展树

509 作为最后一个例子，我们来分析伸展树的运行时间。由第4章得知，在对某项 $X$ 进行访问之后，一步伸展通过下述三种一系列的树操作将 $X$ 移至根处：单旋转（zig）、之字形旋转（zig-zag）和一字形旋转（zig-zig）。树的这些旋转如图11-21所示。我们约定：如果在结点 $X$ 执行一次树的旋转，那么旋转前 $P$ 是它的父结点， $G$ 是它的祖父结点（若 $X$ 不是根的儿子们的话）。

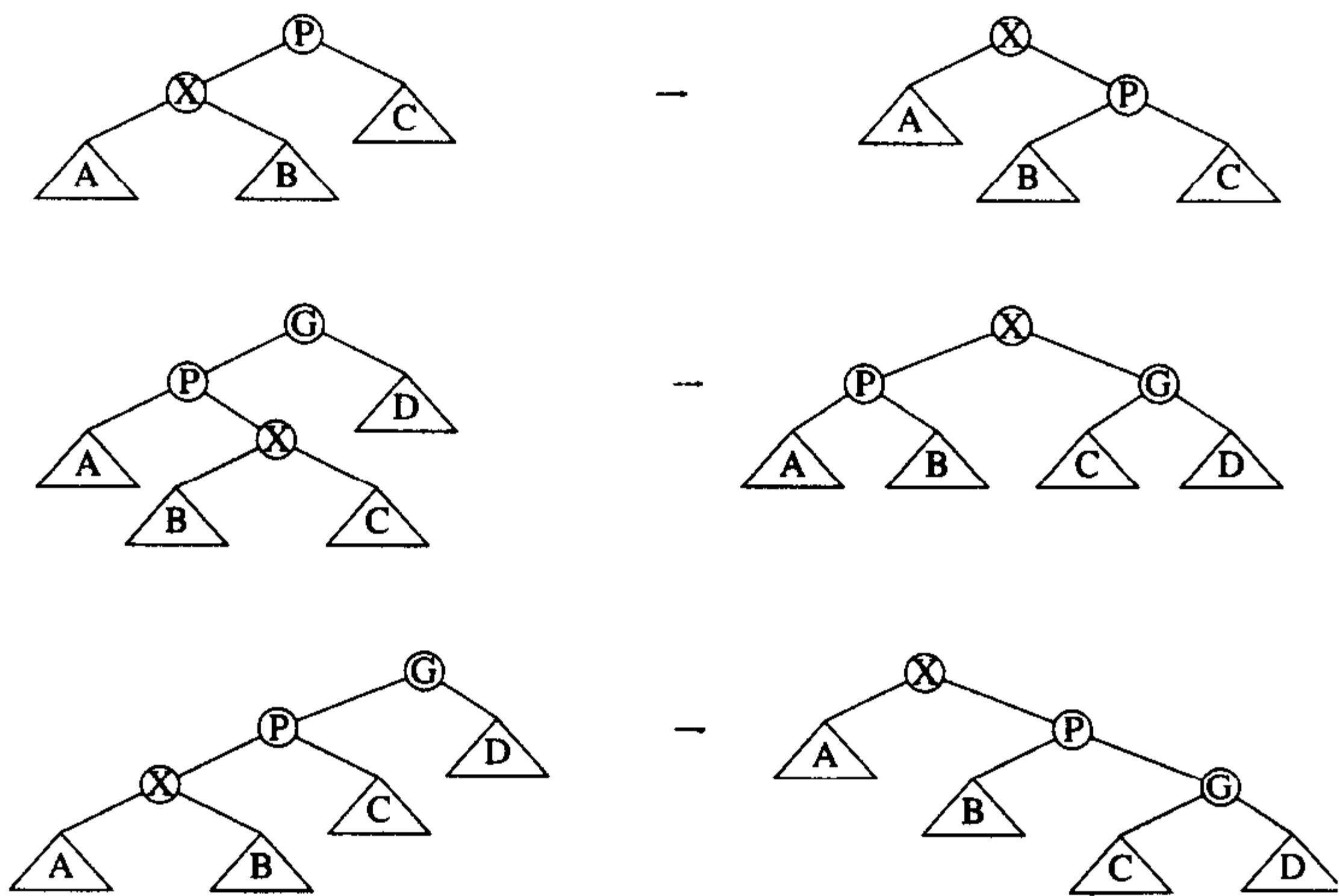


图11-21 单旋转、之字形和一字形旋转操作；每个都有一个对称的情形（未示出）

我们知道，对结点 $X$ 任意的树操作所需的时间正比于从根到 $X$ 的路径上的结点的个数。如果

把每个单旋转操作计为一次旋转，把每个之字形操作或一字形操作计为两次旋转，那么任何访问的花费等于1加上旋转的次数。

为了证明伸展操作的 $O(\log N)$ 摊还时间界，需要一个位势函数，该函数对整个伸展操作最多增加 $O(\log N)$ ，而且在操作期间消去所执行的旋转的次数。找出满足这些原则的位势函数根本不是一件容易的事情。首先容易猜到的位势函数或许就是树上所有结点的深度的和。这个猜测行不通，因为位势在一次访问期间可能增加 $\Theta(N)$ 。当一些元素顺序插入时会有这样的典型例子发生。

一个确实有效的位势函数 $\Phi$ 定义为

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

其中 $S(i)$ 代表 $i$ 的后裔的个数(包括 $i$ 自身)。这个位势函数是对树 $T$ 的所有结点 $i$ 所取的 $S(i)$ 的对数和。

为简化记法，定义

$$R(i) = \log S(i)$$

这使得

$$\Phi(T) = \sum_{i \in T} R(i)$$

$R(i)$ 代表结点 $i$ 的秩。这个术语类似于不相交集算法、二项队列和斐波那契堆的分析中所使用的术语。在所有这些数据结构中，秩的意义多少有些不同，不过，一般是指树的大小的对数的阶(幅度)。对于具有 $N$ 个结点的一棵树 $T$ ，根的秩就是 $R(T) = \log N$ 。用秩的和作为位势函数类似于使用高度的和作为位势函数。重要的差别在于，当一次旋转可以改变树中许多结点的高度时，却只有 $X$ 、 $P$ 和 $G$ 的秩发生变化。

在证明主要的定理之前，我们需要下面的引理。

**引理11.4** 如果 $a + b \leq c$ ，且 $a$ 和 $b$ 均为正整数，那么 $\log a + \log b \leq 2\log c - 2$ 。

**证明** 根据算术-几何平均不等式，

$$\sqrt{ab} \leq (a + b)/2$$

于是

$$\sqrt{ab} \leq c/2$$

510

两边平方得到

$$ab \leq c^2/4$$

两边再取对数，则定理得证。 ■

下面就来证明主要定理，证明过程中注意所用到的一些预备知识。

**定理11.5** 在结点 $X$ 伸展一棵根为 $T$ 的树的摊还时间最多为 $3(R(T) - R(X)) + 1 = O(\log N)$ 。

**证明** 位势函数取为 $T$ 中结点的秩的和。

如果 $X$ 是 $T$ 的根，那么不存在旋转，因此位势没有变化。访问该结点的时间是1；于是，摊还时间为1，定理成立。因此，可以假设至少有一次旋转。

对于任意一步伸展操作，令 $R_i(X)$ 和 $S_i(X)$ 是在这步操作前 $X$ 的秩和大小，并令 $R_f(X)$ 和 $S_f(X)$ 是在这步伸展操作后 $X$ 的秩和大小。我们将证明对一次单旋转所需要的摊还时间最多为 $3(R_f(X) - R_i(X)) + 1$ ，而对一次之字形旋转或一字形旋转的摊还时间最多为 $3(R_f(X) - R_i(X))$ 。我们将证明，当对所有各步伸展求和时，所得到的和就是想要的时间界。

**一步单旋转：**对于单旋转，实际时间为1，而位势变化为 $R_f(X) + R_f(P) - R_i(X) - R_i(P)$ 。注意，位势变化容易计算，因为只有 $X$ 的和 $P$ 的树大小有变化。于是

$$AT_{\text{zig}} = 1 + R_f(X) + R_f(P) - R_i(X) - R_i(P)$$

从图11-21可以看到 $S_i(P) \geq S_f(P)$ ，因此得到 $R_i(P) \geq R_f(P)$ 。这样，

$$AT_{\text{zig}} \leq 1 + R_f(X) - R_i(X)$$

由于 $S_f(X) \geq S_i(X)$ ，于是 $R_f(X) - R_i(X) \geq 0$ ，因此可以增加右边，得到

$$AT_{\text{zig}} \leq 1 + 3(R_f(X) - R_i(X))$$

一步之字形旋转：对于这种情况，实际的花费是2，而位势变化为 $R_f(X) + R_f(P) + R_f(G) - R_i(X) - R_i(P) - R_i(G)$ 。这就给出一个摊还时间界：

$$AT_{\text{zig-zag}} = 2 + R_f(X) + R_f(P) + R_f(G) - R_i(X) - R_i(P) - R_i(G)$$

从图11-21可以看到， $S_f(X) = S_i(G)$ ，于是它们的秩必然相等。因此得到

$$AT_{\text{zig-zag}} = 2 + R_f(P) + R_f(G) - R_i(X) - R_i(P)$$

我们还看到 $S_i(P) \geq S_i(X)$ ，因而 $R_i(X) \leq R_i(P)$ 。代入右边得到

511

$$AT_{\text{zig-zag}} \leq 2 + R_f(P) + R_f(G) - 2R_i(X)$$

从图11-21可以看到， $S_f(P) + S_f(G) \leq S_f(X)$ 。如果应用引理11.4，那么得到

$$\log S_f(P) + \log S_f(G) \leq 2 \log S_f(X) - 2$$

由秩的定义可知，它变成

$$R_f(P) + R_f(G) \leq 2 R_f(X) - 2$$

将其代入，则得

$$\begin{aligned} AT_{\text{zig-zag}} &\leq 2R_f(X) - 2R_i(X) \\ &\leq 2(R_f(X) - R_i(X)) \end{aligned}$$

由于 $R_f(X) \geq R_i(X)$ ，因此得到

$$AT_{\text{zig-zag}} \leq 3(R_f(X) - R_i(X))$$

一步一字形旋转：第三种情况是一字形旋转。这种情形的证明非常类似于之字形。重要的不等式是 $R_f(X) = R_i(G)$ ， $R_f(X) \geq R_f(P)$ ， $R_i(X) \leq R_i(P)$ ，以及 $S_i(X) + S_f(G) \leq S_f(X)$ 。我们把具体细节留作练习11.8。

整个伸展的摊还开销是各步伸展的摊还开销的和。图11-22显示了在结点2的伸展中所执行的各项伸展的过程。令 $R_1(2)$ 、 $R_2(2)$ 、 $R_3(2)$ 和 $R_4(2)$ 分别是这4棵树在结点2的秩。第一步是之字形旋转，其开销最多为 $3(R_2(2) - R_1(2))$ 。第二步是一字形旋转，其开销为 $3(R_3(2) - R_2(2))$ 。最后一步是单旋转，开销不超过 $3(R_4(2) - R_3(2)) + 1$ 。因此总的花费是 $3(R_4(2) - R_1(2)) + 1$ 。

一般地，通过把所有旋转（其中最多有一个是单旋转）的摊还时间加起来，可以看到，在结点 $X$ 伸展的总的摊还时间最多为 $3(R_f(X) - R_i(X)) + 1$ ，其中 $R_i(X)$ 是 $X$ 在第一步伸展前的秩，而 $R_f(X)$ 是 $X$ 在最后一步伸展后的秩。由于最后一次伸展把 $X$ 留在根处，因此得到 $3(R_f(T) - R_i(X)) + 1$ 的摊还界，这个界为 $O(\log N)$ 。 ■

512

因为对伸展树的每一次操作都需要一次伸展，因此任意操作的摊还时间在一次伸展的摊还时间的常数倍数之内。因此，所有伸展树操作花费 $O(\log N)$ 摊还时间。通过使用更一般的位势函数，可以证明伸展树具有若干显著的性质。更多的细节在练习中讨论。

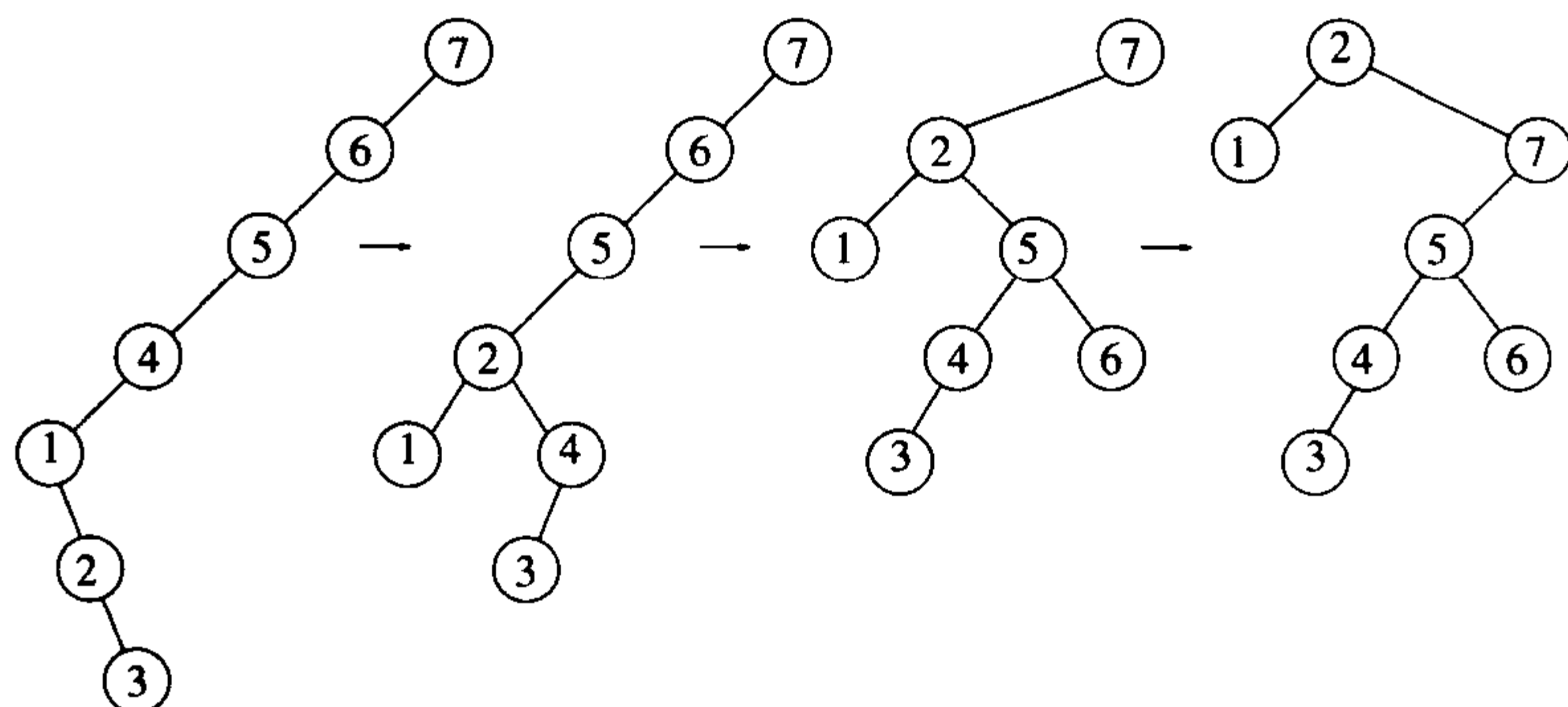


图11-22 在结点2的伸展中涉及的伸展步骤

## 小结

本章讨论了摊还分析如何在一些操作之间分配负荷。为了进行分析，构造一个虚构的位势函数，这个位势函数度量系统的状态。高位势的数据结构是易变的，它建立在相对低廉的操作之上。当操作需要昂贵的开销时，它会由前面一些操作的节省来支付。可以把位势看成是对付灾难的潜能，因为非常昂贵的操作只有在数据结构具有高位势以及已经使用的时间比规定的时间少很多时才可能发生。

数据结构中的低位势意味着每次操作的开销大致等于指定给它的消耗量。负位势意味着欠债；开销的时间多于规定的时间，因此分配（或摊还）的时间不是一个有意义的界。

正如式（11.2）所表达的，一次操作的摊还时间等于实际时间和位势变化的和。整个操作序列的摊还时间等于总的序列操作时间加上位势的净变化。只要这个净变化是正的，那么摊还界就提供实际时间开销的一个上界并且是有意义的。

选择位势函数的关键在于，保证最小的位势要产生在算法的开始，并使得位势对低廉的操作增加而对高昂的操作减少。重要的是过剩或节省的时间要由位势中相反的变化来度量。但是，有时候说起来容易做起来难。

## 练习

- 11.1 什么时候向一个二项队列进行连续 $M$ 次插入开销少于 $2M$ 个时间单元的时间？
- 11.2 设建立一个有 $N = 2^k - 1$ 个元素的二项队列，交替进行 $M$ 对insert和deleteMin操作。显然，每次操作花费 $O(\log N)$ 时间。为什么这与插入的 $O(1)$ 摊还时间界不矛盾？
- \*11.3 通过给出一系列导致一次merge需要 $\Theta(N)$ 时间的操作，证明对于斜堆操作的 $O(\log N)$ 摊还界不能转换成最坏情形界。
- \*11.4 指出如何以一趟自顶向下来合并两个斜堆，并将merge的开销减到 $O(1)$ 摊还时间。
- 11.5 扩展斜堆以支持具有 $O(\log N)$ 摊还时间的decreaseKey操作。
- 11.6 实现斐波那契堆并比较其与二叉堆在用于Dijkstra算法时的性能。
- 11.7 斐波那契堆的标准实现方法需要每个结点4个链（父亲、儿子以及2个兄弟）。指出如何减少链的数量而运行时间开销最多是一个常数因子。
- 11.8 证明：一次一字形伸展的摊还时间最多为 $3(R_f(X) - R_i(X))$ 。
- 11.9 通过改变位势函数能够证明伸展的不同的界。令权函数（weight function） $W(i)$ 为指定给树中每个



结点的某个函数，令  $S(i)$  为以  $i$  为根的子树上所有结点（包括结点  $i$  本身）的权的和。对于与用在伸展界的证明中的该函数相对应的所有结点，特殊情况为  $W(i) = 1$ 。令  $N$  为树中结点的个数，并令  $M$  为访问的次数。证明下列两个定理：

- a. 总的访问时间是  $O(M + (M + N) \log N)$ 。
- \*b. 如果  $q_i$  为项  $i$  被访问的次数，而对所有的  $i$ ， $q_i > 0$ ，那么总的访问时间为

$$O\left(M + \sum_{i=1}^N q_i \log(M / q_i)\right)$$

- 11.10 a. 指出如何实现对伸展树的 merge 操作使得从  $N$  个单元素树开始的任意  $N-1$  次 merge 操作序列开销  $O(N \log^2 N)$  时间。

\*b. 将这个界改进为  $O(M \log N)$ 。

- 11.11 第 5 章描述了再散列 (rehashing)：当一个表的表元素超过容量一半的时候，则构造一个两倍大的新表，而整个老表要被再散列。使用位势函数给出一个正式的摊还分析来证明一次插入操作的摊还时间仍为  $O(1)$ 。

- 11.12 斐波那契堆的最大深度是多少？

- 11.13 具有堆序的双端队列 (deque) 是由一些项的表组成的数据结构，可以对其进行下列操作：

push(x)：将项  $x$  插入到双端队列的前端。

pop()：从双端队列中除去前端项并将它返回。

inject(x)：把项  $x$  插入到双端队列的尾端。

eject()：从双端队列中除去尾端项并将它返回。

findMin()：返回双端队列的最小项。

a. 描述如何以每次操作对应常数摊还时间支持这些操作。

\*\*b. 描述如何以每次操作常数最坏情形时间支持这些操作。

- 11.14 证明二项队列实际上以  $O(1)$  摊还时间支持合并操作。定义二项队列的位势为树的棵数加上最大的树的秩。

- 11.15 设为了节省时间，把伸展对每隔一次树操作进行。摊还的开销还是对数的吗？

- 11.16 在伸展树的界的证明中使用位势函数，伸展树的最大位势和最小位势是什么？在一次伸展中，位势函数可以减少多少？在一次伸展中，位势函数可以增加多少？可以给出大  $O$  解。

- 11.17 作为伸展的结果，访问路径上的大部分结点朝根的方向移动一半距离，而该路径上的少数几个结点却向下移动一层。这就提出使用每个结点的深度的对数的和作为位势函数。

a. 位势函数的最大值是多少？

b. 位势函数的最小值是多少？

c. (a) 问和 (b) 问的答案的差给出某种提示，即该位势函数不是太好。证明，一次伸展操作可能以  $\Theta(N/\log N)$  增加位势。

514

## 参考文献

论文[10]提供了对摊还分析的极好的综述。

下面的参考文献中有许多和前几章中的相同，这里再次引用它们是为了方便和完善。二项队列首先在[11]中阐述并在[1]中分析。练习11.3和11.4的解法见于论文[9]。斐波那契堆在[3]中论述。练习11.9 (a)指出，在最佳静态查找树的一个常数因子之内伸展树是最优的。练习11.9 (b)则指出，伸展树在最佳最优查找树的一个常数因子之内是最优的。这些以及另外两个强结果在原始的伸展树论文[7]中得以证明。

[2]中使用摊还来有效地合并平衡查找树。伸展树的merge操作在[6]中描述。练习11.13的一种解法可在[4]中找到。练习11.14取自文献[5]。

在[8]中使用摊还分析设计一种联机算法, 该算法处理一系列查询, 其所花费的时间比同类问题的脱机算法只多一个常数因子。

1. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms," *SIAM Journal on Computing*, 7 (1978), 298-319.
2. M. R. Brown and R. E. Tarjan, "Design and Analysis of a Data Structure for Representing Sorted Lists," *SIAM Journal on Computing*, 9 (1980), 594-614.
3. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM*, 34 (1987), 596-615.
4. H. Gajewska and R. E. Tarjan, "Dequeues with Heap Order," *Information Processing Letters*, 22 (1986), 197-200.
5. C. M. Khoong and H. W. Leong, "Double-Ended Binomial Queues," *Proceedings of the Fourth Annual International Symposium on Algorithms and Computation* (1993), 128-137.
6. G. Port and A. Moffat, "A Fast Algorithm for Melding Splay Trees," *Proceedings of First Workshop on Algorithms and Data Structures* (1989), 450-459. 515
7. D. D. Sleator and R. E. Tarjan, "Self-adjusting Binary Search Trees," *Journal of the ACM*, 32 (1985), 652-686.
8. D. D. Sleator and R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM*, 28 (1985), 202-208.
9. D. D. Sleator and R. E. Tarjan, "Self-adjusting Heaps," *SIAM Journal on Computing*, 15 (1986), 52-69.
10. R. E. Tarjan, "Amortized Computational Complexity," *SIAM Journal on Algebraic and Discrete Methods*, 6 (1985), 306-318.
11. J. Vuillemin, "A Data Structure for Manipulating Priority Queues," *Communications of the ACM*, 21 (1978), 309-314. 516

**本**章讨论7种强调实用的数据结构。首先考察第4章讨论过的AVL树的变种，包括优化的伸展树、红黑树、（第10章讨论过的）跳跃表的确定性的形式、AA树以及treap树。

然后考察一种可以用于多维数据的数据结构。在这种情况下，每一项均可有多个键。 $k$ -d树对任何键都能进行相关的搜索。

最后，考察配对堆（pairing heap），它似乎是斐波那契堆最实用的变体。

重现的论题包括：

- 在适当的时候非递归的自顶向下（而不是自底向上）的查找树的各种实现方法。
- 详细的、优化的尤其是利用标记结点的实现方法。

### 12.1 自顶向下伸展树

第4章讨论了基本的伸展树操作。当项 $X$ 作为树叶被插入时，称为伸展（splay）的一系列的树旋转使得 $X$ 成为树的新根。伸展操作也在查找期间执行，而且如果一项也没有找到，那么就要对访问路径上的最后的结点施行一次伸展。第11章指出一次伸展树操作的摊还开销为 $O(\log N)$ 。

这种伸展操作的直接实现需要从根沿树往下的一次遍历，以及而后的自底向上的一次遍历。这可以通过保存一些父链来完成，也可以通过将访问路径存储到栈中来完成。但是，这两种方法均需大量的开销，而且二者都必须处理许多特殊的情况。在这一节，我们指出如何在初始访问路径上施行一些旋转。结果是得到在实践中更快的过程，只用到 $O(1)$ 的附加空间，但却保持了 $O(\log N)$ 的摊还时间界。

**517** 图12-1指出单旋转、一字形旋转和之字形旋转（照惯例，忽略三种对称的旋转）。在访问的任一时刻，都有一个当前结点 $X$ ，它是其子树的根；在图12-1中它被表示成“中间”的树<sup>1</sup>。树 $L$ 存储树 $T$ 中小于 $X$ 的结点，但不存储 $X$ 的子树的结点；类似地，树 $R$ 存储大于 $X$ 的结点，但不存储 $X$ 的子树的结点。初始时 $X$ 为 $T$ 的根，而 $L$ 和 $R$ 是空树。

如果旋转是一次单旋转，那么根在 $Y$ 的树变成中间树的新根。 $X$ 和子树 $B$ 作为 $R$ 中最小项的左儿子附接到 $R$ 上； $X$ 的左儿子逻辑上成为null<sup>2</sup>。结果， $X$ 成为 $R$ 的新的最小项。特别要注意，为使单旋转情形适用， $Y$ 不一定必须是树叶。如果查找小于 $Y$ 的项，而 $Y$ 没有左儿子（但确实有一个右儿子），那么这种单旋转情形将是适用的。

1. 为简单起见，我们不区分一个“结点”和该结点中的项。

2. 在程序中 $R$ 的最小结点没有null左链，因为没有必要。这意味着，`printTree(r)`将包含某些项，这些项逻辑上不在 $R$ 中。

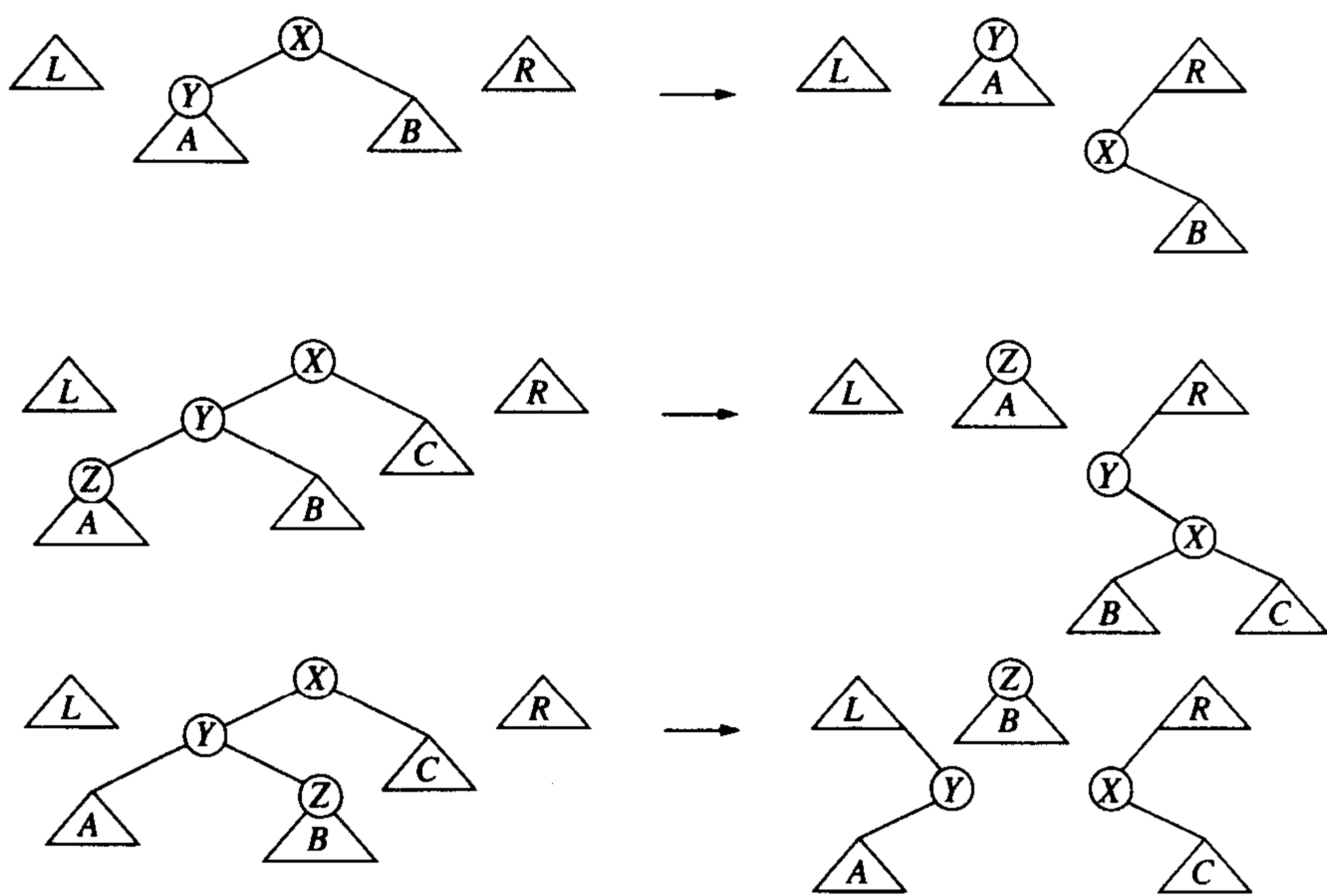


图12-1 自顶向下伸展旋转：单旋转、一字形旋转和之字形旋转

对于一字形旋转情形，有类似的剖析。关键是在 $X$ 和 $Y$ 之间施行的一次旋转。之字形旋转的情形把底部结点 $Z$ 带到中间树的顶部，并把子树 $X$ 和 $Y$ 分别附接到 $R$ 和 $L$ 上。注意， $Y$ 被附接后成为 $L$ 中的最大项。

之字形旋转步骤多少可以得到简化，因为没有旋转要执行， $Z$ 不再是中间树的根， $Y$ 成为该根，如图12-2所示。因为之字形情形的动作变成与单旋转情形的相同，所以编程得到简化。看起来这是有利的，因为对大量情形的测试是要消耗时间的。其缺点在于，为了仅仅降低一层，在伸展过程中却要进行更多的迭代。

518

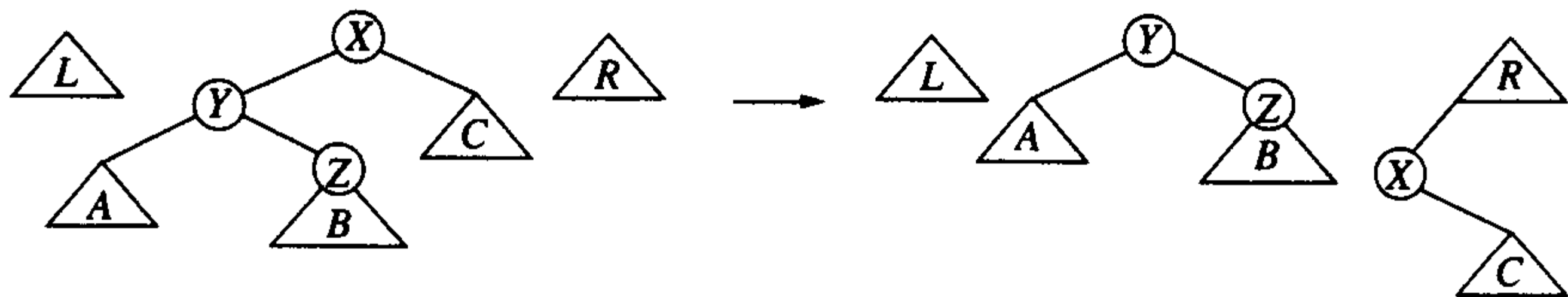


图12-2 简化的自顶向下的之字形旋转

图12-3指出一旦执行完最后一步伸展，将如何处理 $L$ 、 $R$ 和中间树以形成一棵树。特别要注意，这里的结果不同于自底向上的伸展。关键的问题在于它保持了 $O(\log N)$ 的摊还界（练习12.1）。

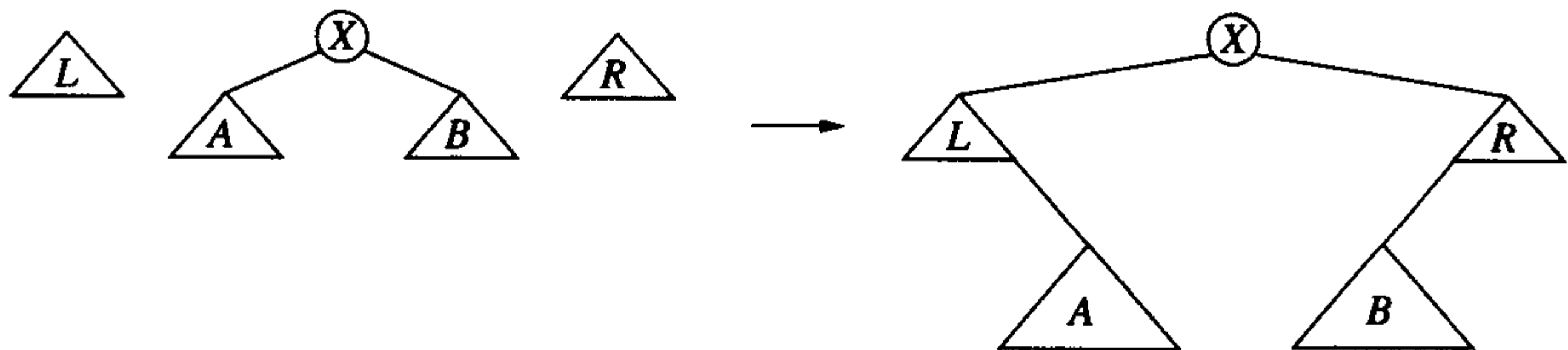


图12-3 自顶向下伸展的最后整理

自顶向下伸展算法的一个例子如图12-4所示。我们想要访问树中的19。第一步是一个之字形



旋转。根据图12-2（的对称形式），把根在25的子树带到中间树的根处，并把12和它的左子树附接到L上。

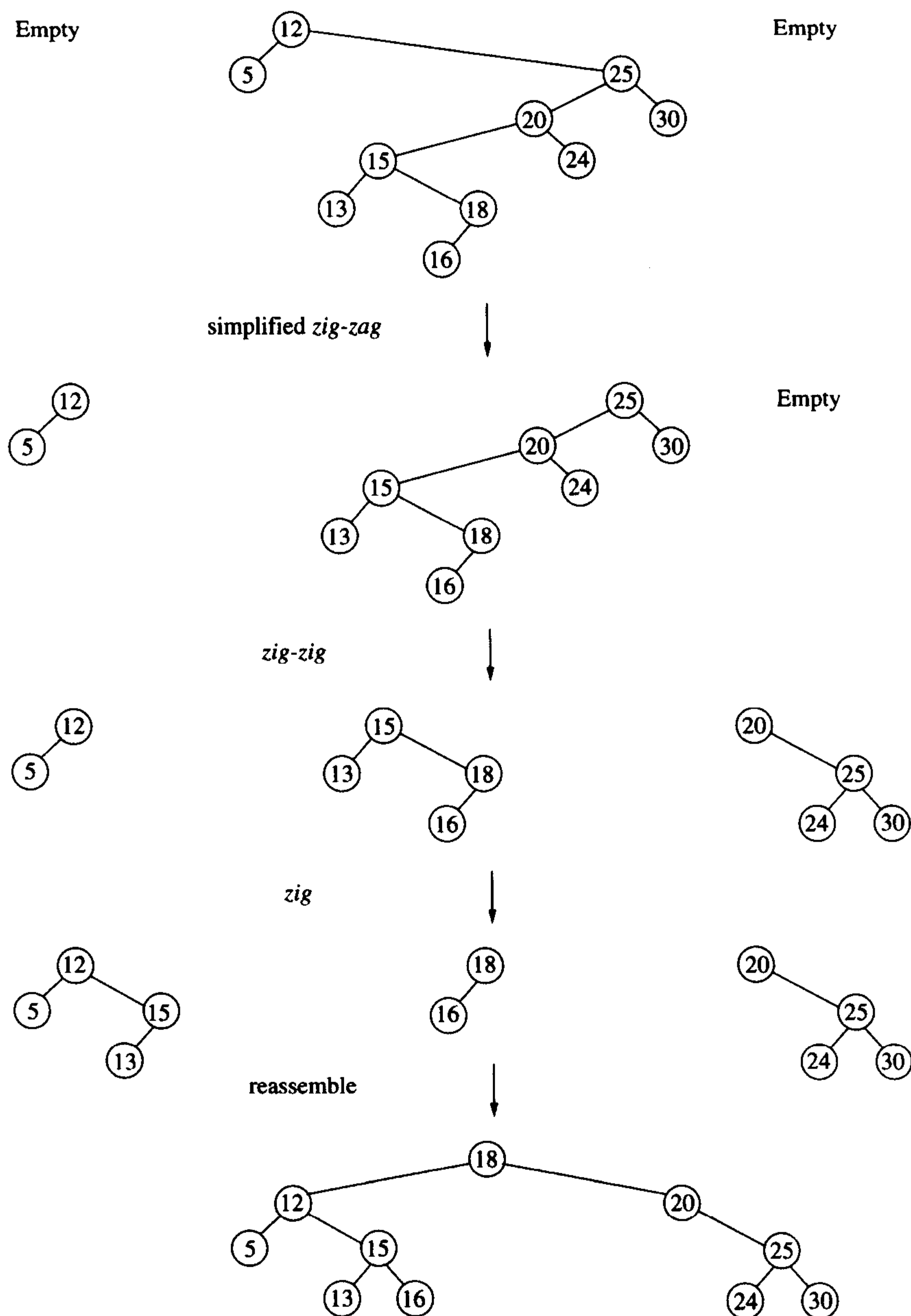


图12-4 自顶向下伸展的各步（访问上面树中的19）

下一步是一个一字形旋转：15被提高到中间树的根处，并在20和25之间进行一次旋转，所得到的子树被附接到R上。此时查找19导致最后的单旋转。中间树的新根为18，而15和它的左子树作为L的最大结点的右儿子被接上。根据图12-3重新组装则该步伸展结束。

使用带有左链和右链的一个头结点最终引用左树的根和右树的根。由于这两棵树初始为空，

因此使用一个头结点分别对应初始状态右树或左树的最小或最大结点。这种方法可以使得程序避免检测空树。第一次左树变成非空时，右链将被初始化并在以后保持不变；这样，在自顶向下查找的最后它将包含左树的根。类似地，左链最终将包含右树的根。

SplayTree类接口及其构造函数和析构函数如图12-5所示。构造函数分配nullNode标记。使用标记nullNode逻辑上表示一个NULL指针。析构函数在调用makeEmpty之后将其delete。我们反复使用这种技术来简化程序（因而使得程序多少要快一些）。图12-6给出伸展过程的程序，这里的header结点使我们肯定能够把X附接到R的最大结点上，而不必担心R可能是空的（对于处理L的对称情形类似地进行）。

```

1  template <typename Comparable>
2  class SplayTree
3  {
4  public:
5      SplayTree( )
6      {
7          nullNode = new BinaryNode;
8          nullNode->left = nullNode->right = nullNode;
9          root = nullNode;
10     }
11
12     ~SplayTree( )
13     {
14         makeEmpty( );
15         delete nullNode;
16     }
17
18     // Same methods as for BinarySearchTree (omitted)
19
20     SplayTree( const SplayTree & rhs );
21     const SplayTree & operator=( const SplayTree & rhs );
22
23 private:
24     struct BinaryNode
25     { /* Usual code for binary search tree nodes */ };
26
27     BinaryNode *root;
28     BinaryNode *nullNode;
29
30     // Same methods as for BinarySearchTree (omitted)
31
32     // Tree manipulations
33     void rotateWithLeftChild( BinaryNode * & k2 );
34     void rotateWithRightChild( BinaryNode * & k1 );
35     void splay( const Comparable & x, BinaryNode * & t );
36 };

```

图12-5 伸展树：类接口、构造函数和析构函数

正如前面所提到的，在伸展到最后重新组装之前，header.left和header.right分别引用R和L的根（这不是排印错误，而是遵守链的指向）。除了这个细节之外，该程序是相对简单的。

图12-7显示了将一项插入到一棵树中的方法。一个新的结点被分配（如果需要），且如果树是空的，那么就建立一棵单结点树；否则，围绕被插入的值x伸展root。若新根上的数据等于x，则出现一个重复元；不是再次插入x，而是为将来的插入保留newNode并立即返回。如果新根包

519

520  
522

523

newNode的左子树。如果root的新根包含有小于x的值，那么类似的逻辑仍然适用。在这两种情况下，newNode均成为新的根。

```

1  /**
2   * Internal method to perform a top-down splay.
3   * The last accessed node becomes the new root.
4   * This method may be overridden to use a different
5   * splaying algorithm, however, the splay tree code
6   * depends on the accessed item going to the root.
7   * x is the target item to splay around.
8   * t is the root of the subtree to splay.
9   */
10 void splay( const Comparable & x, BinaryNode * & t )
11 {
12     BinaryNode *leftTreeMax, *rightTreeMin;
13     static BinaryNode header;
14
15     header.left = header.right = nullNode;
16     leftTreeMax = rightTreeMin = &header;
17
18     nullNode->element = x; // Guarantee a match
19
20     for( ; ; )
21         if( x < t->element )
22         {
23             if( x < t->left->element )
24                 rotateWithLeftChild( t );
25             if( t->left == nullNode )
26                 break;
27             // Link Right
28             rightTreeMin->left = t;
29             rightTreeMin = t;
30             t = t->left;
31         }
32         else if( t->element < x )
33         {
34             if( t->right->element < x )
35                 rotateWithRightChild( t );
36             if( t->right == nullNode )
37                 break;
38             // Link Left
39             leftTreeMax->right = t;
40             leftTreeMax = t;
41             t = t->right;
42         }
43         else
44             break;
45
46     leftTreeMax->right = t->left;
47     rightTreeMin->left = t->right;
48     t->left = header.right;
49     t->right = header.left;
50 }

```

图12-6 自顶向下伸展方法

第4章已经证明了伸展树中的删除是很容易的，因为一次伸展即把删除目标置于根处。最后我们给出图12-8中的删除例程。删除过程比对应的插入过程还要短，确实罕见。图12-8还给出了makeEmpty。简单的递归后序遍历回收树结点是不安全的，因为伸展树即使性能很好，仍有可能

极度不平衡。在这种情况下，递归可能会导致溢出。这里使用了一个简单的替代办法，仍旧是  $O(N)$  的（虽然这并不明显）。对 `operator=` 也需要类似的考虑。

```

1 void insert( const Comparable & x )
2 {
3     static BinaryNode *newNode = NULL;
4
5     if( newNode == NULL )
6         newNode = new BinaryNode;
7     newNode->element = x;
8
9     if( root == nullNode )
10    {
11        newNode->left = newNode->right = nullNode;
12        root = newNode;
13    }
14    else
15    {
16        splay( x, root );
17        if( x < root->element )
18        {
19            newNode->left = root->left;
20            newNode->right = root;
21            root->left = nullNode;
22            root = newNode;
23        }
24        else
25            if( root->element < x )
26            {
27                newNode->right = root->right;
28                newNode->left = root;
29                root->right = nullNode;
30                root = newNode;
31            }
32        else
33            return;
34    }
35    newNode = NULL; // So next insert will call new
36 }

```

图12-7 自顶向下伸展树的insert

## 12.2 红黑树

历史上流行的AVL树的另一变种是红黑树（red black tree）。对红黑树的操作在最坏情形下花费  $O(\log N)$  时间，而且我们将看到，对于插入操作的一种慎重的非递归实现可以相对容易地完成（与AVL树相比）。

红黑树是具有下列着色性质的二叉查找树：

- (1) 每一个结点或者着红色，或者着黑色。
- (2) 根是黑色的。
- (3) 如果一个结点是红色的，那么它的子结点必须是黑色的。
- (4) 从一个结点到一个NULL指针的每一条路径都必须包含相同数目的黑色结点。

着色规则的一个结论是，红黑树的高度最多是  $2 \log(N+1)$ 。因此，查找保证是一种对数的操作。图12-9显示了一棵红黑树，其中的红色结点用双圆圈表示。



```

1 void remove( const Comparable & x )
2 {
3     BinaryNode *newTree;
4
5     // If x is found, it will be at the root
6     splay( x, root );
7     if( root->element != x )
8         return; // Item not found; do nothing
9
10    if( root->left == nullNode )
11        newTree = root->right;
12    else
13    {
14        // Find the maximum in the left subtree
15        // Splay it to the root; and then attach right child
16        newTree = root->left;
17        splay( x, newTree );
18        newTree->right = root->right;
19    }
20    delete root;
21    root = newTree;
22 }
23
24 void makeEmpty( )
25 {
26     while( !isEmpty( ) )
27     {
28         findMax( ); // Splay max item to root
29         remove( root->element );
30     }
31 }

```

图12-8 自顶向下的删除过程和makeEmpty

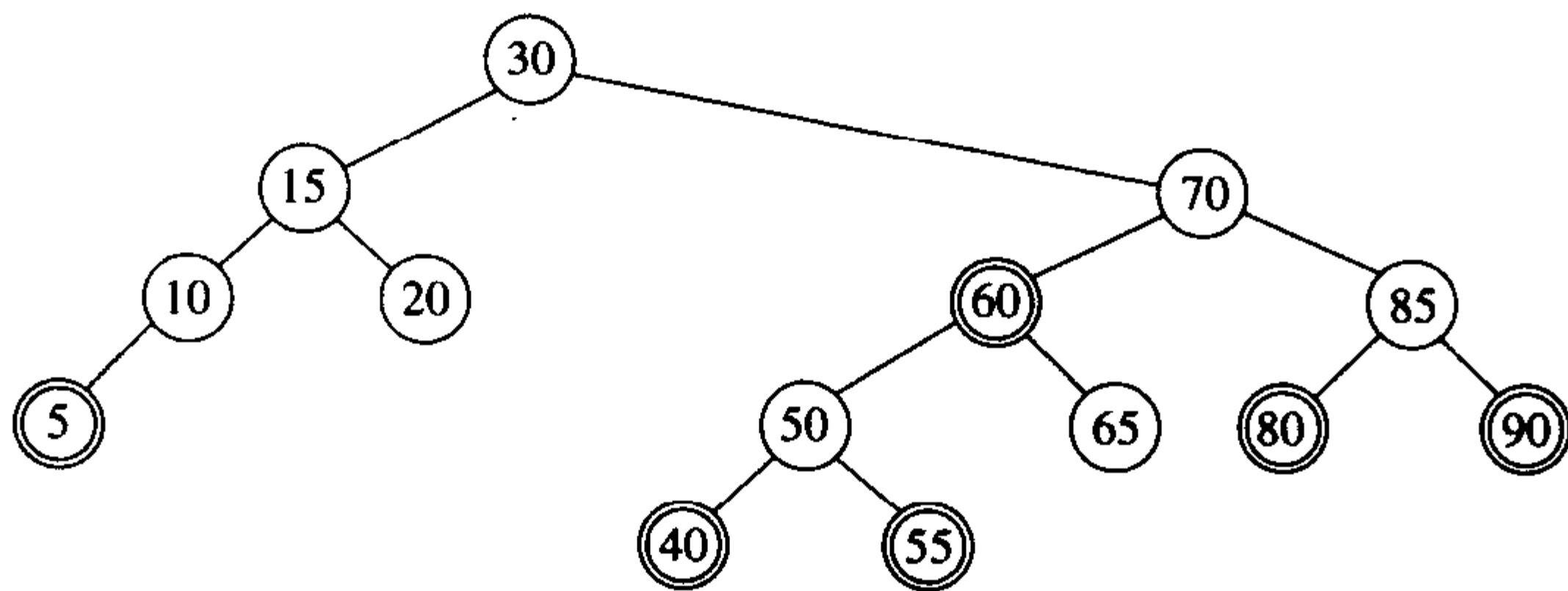


图12-9 红黑树的例子（插入序列为：10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55）

和通常一样，困难在于将一个新项插入到树中。通常把新项作为树叶放到树中。如果把该项涂成黑色，那么肯定违反红黑树的着色性质4，因为将会建立一条更长的黑色结点的路径。因此，这一项必须涂成红色。如果它的父结点是黑色的，则插入完成。如果它的父结点已经是红色的，那么得到连续的两个红色结点，这就违反了红黑树的着色性质3。在这种情况下，必须调整该树以确保满足红黑树的着色性质3（且又不违反红黑树的着色性质4）。用于完成这项任务的基本操作是颜色的改变和树的旋转。

525

### 12.2.1 自底向上插入

我们已经提到，如果新插入的项的父结点是黑色的，那么插入完成。因此，将25插入到图12-9的树中是简单的操作。

如果父结点是红色的，那么有几种情形（每种都有一个镜像对称）需要考虑。首先，假设这个父结点的兄弟是黑色的（我们采纳约定：NULL结点都是黑色的）。这对于插入3或8是适用的，但对插入99不适用。令 $X$ 是新加的树叶， $P$ 是它的父结点， $S$ 是该父结点的兄弟结点（若存在）， $G$ 是祖父结点。在这种情形下只有 $X$ 和 $P$ 是红色的， $G$ 是黑色的，因为否则就会在插入前有两个相连的红色结点，违反了红黑树的着色性质。采用伸展树的术语， $X$ 、 $P$ 和 $G$ 可以形成一个一字形链或之字形链（两个方向中的任一个方向）。图12-10指出，当 $P$ 是一个左儿子时（注意有一个对称情形）如何旋转该树。即便 $X$ 是一片树叶，我们还是画出了更一般的情形，使得 $X$ 在树的中间，后面将用到这个更一般的旋转。

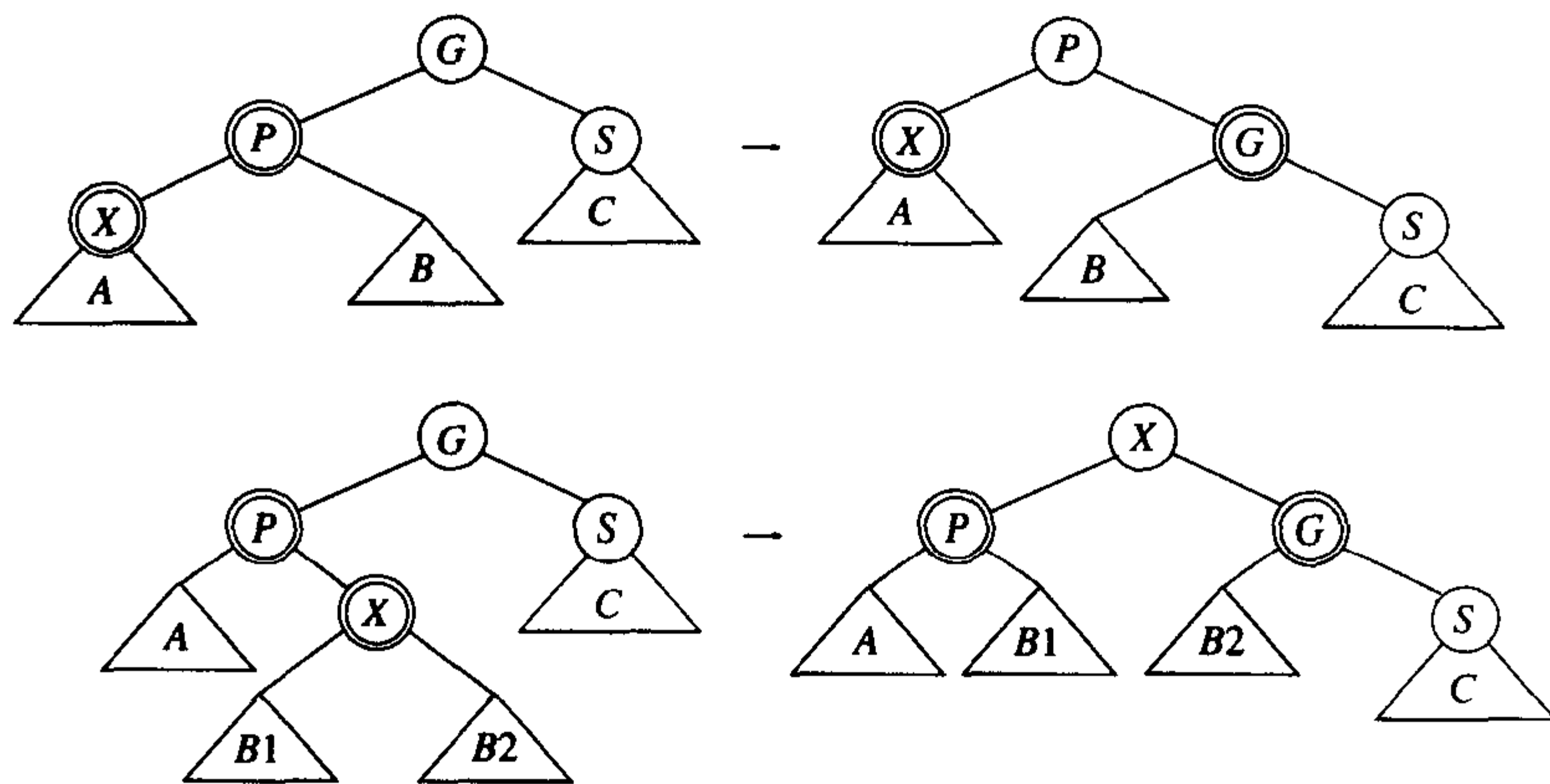


图12-10 如果 $S$ 是黑色的，则单旋转和之字形旋转有效

第一种情形对应 $P$ 和 $G$ 之间的单旋转，而第二种情形对应双旋转，该双旋转首先在 $X$ 和 $P$ 间进行，然后在 $X$ 和 $G$ 之间进行。编写程序的时候，必须记录父结点、祖父结点，以及为了重新连接还要记录曾祖父结点。

在这两种情形下，子树的新根均被涂成黑色，因此，即便原来的曾祖是红色的，也排除了两个相邻红色结点的可能性。同样重要的是，这些旋转的结果使通向 $A$ 、 $B$ 和 $C$ 诸路径上的黑色结点个数保持不变。

到现在为止一切顺利。但是，正如企图将79插入到图12-9树中的情况一样，如果 $S$ 是红色的，那么会发生什么情况呢？在这种情况下，初始时从子树的根到 $C$ 的路径上有一个黑色结点。在旋转之后，一定仍然还是只有一个黑色结点。但在这两种情况下，在通向 $C$ 的路径上都有三个结点（新的根、 $G$ 和 $S$ ）。由于只有一个可能是黑色的，又由于不能有连续的红色结点，于是必须把 $S$ 和子树的新根都涂成红色，而把 $G$ （以及第四个结点）涂成黑色。这很好，可是，如果曾祖父也是红色的那么又会怎样呢？此时，可以将这个过程朝着根的方向上滤，就像对B树和二叉堆所做的那样，直到不再有两个相连的红色结点或者到达根（它将被重新涂成黑色）处为止。

526

### 12.2.2 自顶向下红黑树

上滤的实现需要用一个栈或用一些父链保存路径。我们看到，如果使用一个自顶向下的过程，实际上是对红黑树应用自顶向下保证 $S$ 不会是红色的过程，则伸展树会更有效。

这个过程在概念上是容易的。在向下的过程中当看到结点 $X$ 有两个红色儿子的时候，让 $X$ 成为红色的而让它的两个儿子是黑色的。图12-11显示了这种颜色翻转的现象，只有当 $X$ 的父结点 $P$ 也是红色的时候这种翻转才会违反红黑树的着色性质。但是此时可以应用图12-10中的适当旋转。

如果X的父结点的兄弟结点是红色的会如何呢？这种可能已经被自顶向下过程中的行动排除，因此X的父结点的兄弟结点不可能是红色的！特别地，如果在沿树向下的过程中看到结点Y有两个红儿子，那么可以知道Y的孙子必然是黑色的，由于Y的儿子也要变成黑色的，甚至在发生旋转之后，因此我们将不会看到两层上其他的红色结点。这样，若X的父结点是红色的，则X的父结点的兄弟不可能也是红色的。



图12-11 颜色翻转：只有当X的父结点是红色的时候才继续旋转

例如，假设要将45插入到图12-9中的树上。在沿树向下的过程中，我们看到50有两个红儿子。因此，执行一次颜色翻转，使50为红的，40和55是黑色的。现在50和60都是红色的。在60和70之间执行单旋转，使得60是30的右子树的黑根，而70和50都是红色的。如果我们在路径上看到其他的含有两个红色儿子的结点，那么继续，执行同样的操作。当到达树叶时，把45作为红结点插入，由于父结点是黑色的，因此插入完成。最后得到的树如图12-12所示。

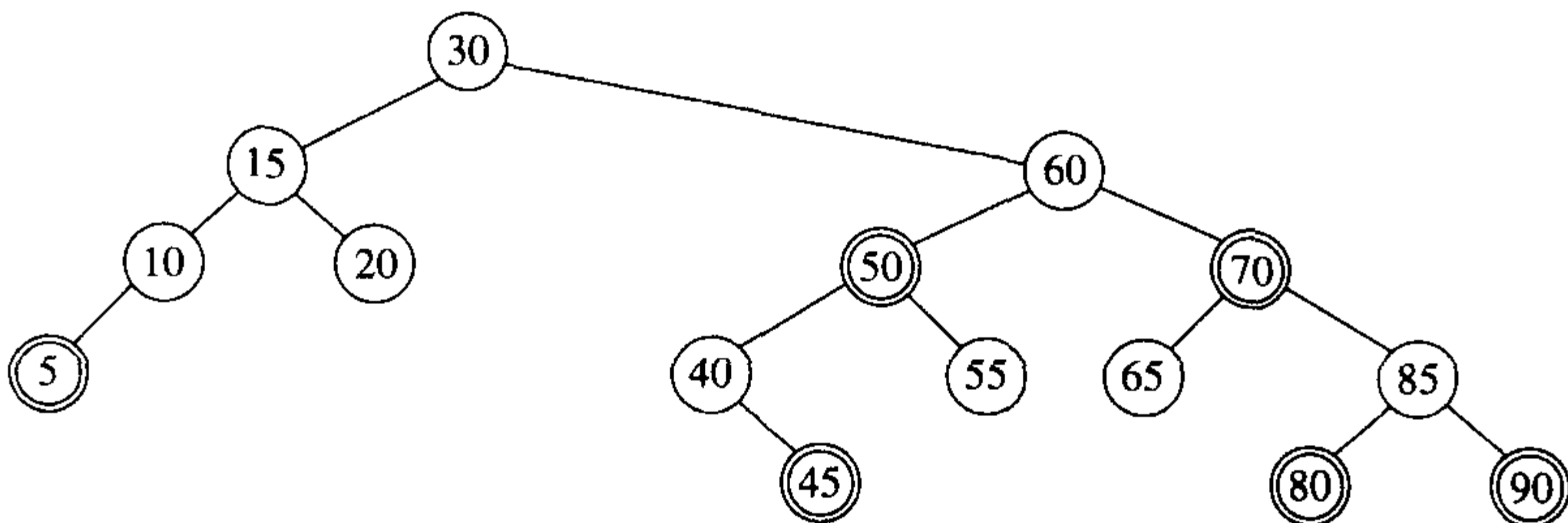


图12-12 将45插入到图12-9中的树上

如图12-12所示，所得的红黑树常常平衡得很好。经验指出，平均红黑树大约和平均AVL树一样深，从而查找时间一般接近最优。红黑树的优点是执行插入所需要的开销相对较低，再有就是实践中发生的旋转相对较少。

527

红黑树的具体实现是很复杂的，这不仅因为可能有大量的旋转，而且还因为一些子树可能是空的（如10的右子树），以及处理根的特殊情况（尤其是根没有父亲）。因此，我们使用两个标记结点：一个是为根，一个是nullNode，其作用像在伸展树中一样是指示一个NULL指针。根标记将存储键 $-\infty$ 和一个指向真正的根的右链。为此，需要调整查找和输出过程。递归的例程都很巧妙。图12-13指出如何重新编写中序遍历。printTree例程的实现很直接。检测 $t \neq t \rightarrow \text{left}$ 可以写成 $t \neq \text{nullNode}$ 。然而，在执行深复制的相似例程里这是一个陷阱。这也在图12-13中示出。在执行别名测试和清空目标树后，operator=调用clone。但是在clone里，测试 $t == \text{nullNode}$ 并不起作用，因为nullNode是目标的nullNode，而不是源的。因此，这里是一个假测试。

图12-14显示了RedBlackTree类框架以及构造函数。

接着，图12-15显示了执行一次单旋转的例程。因为所得到的树必须要附接到父结点上，所以rotate把父结点作为参数。在沿树下行时，我们把item作为参数传递而不是记录旋转的类型。由于期望在插入过程中进行很少的旋转，因此采用这种方式实际上不仅更简单，而且还更快。rotate直接返回执行相应单旋转的结果。

```

1 void printTree( ) const
2 {
3     if( header->right == nullNode )
4         cout << "Empty tree" << endl;
5     else
6         printTree( header->right );
7 }
8
9 void printTree( RedBlackNode *t ) const
10 {
11     if( t != t->left )
12     {
13         printTree( t->left );
14         cout << t->element << endl;
15         printTree( t->right );
16     }
17 }
18
19 const RedBlackTree & operator=( const RedBlackTree & rhs )
20 {
21     if( this != &rhs )
22     {
23         makeEmpty( );
24         header->right = clone( rhs.header->right );
25     }
26
27     return *this;
28 }
29
30 RedBlackNode * clone( RedBlackNode * t ) const
31 {
32     if( t == t->left ) // Cannot test against nullNode!!!
33         return nullNode;
34     else
35         return new RedBlackNode( t->element, clone( t->left ),
36                                   clone( t->right ), t->color );
37 }

```

图12-13 使用printTree和operator=两个标记对树的中序遍历

```

1 template <typename Comparable>
2 class RedBlackTree
3 {
4     public:
5         explicit RedBlackTree( const Comparable & negInf );
6         RedBlackTree( const RedBlackTree & rhs );
7         ~RedBlackTree( );
8
9         const Comparable & findMin( ) const;
10        const Comparable & findMax( ) const;
11        bool contains( const Comparable & x ) const;
12        bool isEmpty( ) const;
13        void printTree( ) const;
14
15        void makeEmpty( );
16        void insert( const Comparable & x );
17        void remove( const Comparable & x );
18
19        enum { RED, BLACK };

```

图12-14 类接口和构造函数



```

20
21     const RedBlackTree & operator=( const RedBlackTree & rhs );
22
23 private:
24     struct RedBlackNode
25     {
26         Comparable    element;
27         RedBlackNode *left;
28         RedBlackNode *right;
29         int            color;
30
31         RedBlackNode( const Comparable & theElement = Comparable( ),
32                     RedBlackNode *lt = NULL, RedBlackNode *rt = NULL,
33                     int c = BLACK )
34             : element( theElement ), left( lt ), right( rt ), color( c ) { }
35     };
36
37     RedBlackNode *header; // The tree header (contains negInf)
38     RedBlackNode *nullNode;
39
40     // Used in insert routine and its helpers (logically static)
41     RedBlackNode *current;
42     RedBlackNode *parent;
43     RedBlackNode *grand;
44     RedBlackNode *great;
45
46     // Usual recursive stuff
47     void reclaimMemory( RedBlackNode *t );
48     void printTree( RedBlackNode *t ) const;
49
50     RedBlackNode * clone( RedBlackNode * t ) const;
51
52     // Red-black tree manipulations
53     void handleReorient( const Comparable & item );
54     RedBlackNode * rotate( const Comparable & item, RedBlackNode *theParent );
55     void rotateWithLeftChild( RedBlackNode * & k2 );
56     void rotateWithRightChild( RedBlackNode * & k1 );
57 };
58
59 /**
60  * Construct the tree.
61  * negInf is a value less than or equal to all others.
62  */
63 explicit RedBlackTree( const Comparable & negInf )
64 {
65     nullNode = new RedBlackNode;
66     nullNode->left = nullNode->right = nullNode;
67     header = new RedBlackNode( negInf );
68     header->left = header->right = nullNode;
69 }

```

图12-14 类接口和构造函数 (续)

最后，我们在图12-16中给出了插入过程。当遇到带有两个红色儿子的结点时调用例程 `handleReorient`，在插入一片树叶时也调用该例程。最为复杂的部分是，一个双旋转实际上是两个单旋转，而且只有当通向x的分支（在 `insert` 方法中由 `current` 表示）取相反方向时才进行。正如在前面的讨论中提到的，当沿树向下进行的时候，`insert` 必须记录父亲、祖父和曾祖父。由于这些量要由 `handleReorient` 共享，因此让它们成为类成员。注意，在一次旋转之后，存储在祖父和曾祖父中的值将不再正确。不过，可以肯定到下一次再需要时它们将被重新存储。

```

1  /**
2   * Internal routine that performs a single or double rotation.
3   * Because the result is attached to the parent, there are four cases.
4   * Called by handleReorient.
5   * item is the item in handleReorient.
6   * theParent is the parent of the root of the rotated subtree.
7   * Return the root of the rotated subtree.
8   */
9  RedBlackNode * rotate( const Comparable & item, RedBlackNode *theParent )
10 {
11     if( item < theParent->element )
12     {
13         item < theParent->left->element ?
14             rotateWithLeftChild( theParent->left ) : // LL
15             rotateWithRightChild( theParent->left ) ; // LR
16         return theParent->left;
17     }
18     else
19     {
20         item < theParent->right->element ?
21             rotateWithLeftChild( theParent->right ) : // RL
22             rotateWithRightChild( theParent->right ) ; // RR
23         return theParent->right;
24     }
25 }

```

图12-15 rotate方法

### 12.2.3 自顶向下删除

红黑树中的删除也可以自顶向下进行。每一件工作都归结于能够删除一个叶结点。这是因为，要删除一个带有两个儿子的结点，可以用右子树上的最小结点代替它；该结点必然最多有一个儿子，然后将该结点删除。只有一个右儿子的结点可以用相同的方式删除，而只有一个左儿子的结点可以通过用其左子树上的最大结点替换而被删除。注意，对于红黑树，对于带有一个儿子的结点的情形，我们不想使用这种方法进行，因为这可能在树的中部连接两个红色结点，使得红黑条件的实现增加困难。

当然，红色树叶的删除很简单。然而，如果一片树叶是黑的，那么删除操作会复杂得多，因为黑色结点的删除将破坏着色性质4。解决方法是保证自顶向下删除期间树叶是红色的。

在整个讨论中，令 $X$ 为当前结点， $T$ 是它的兄弟，而 $P$ 是它们的父亲。开始时把树的根涂成红色。在沿树向下遍历时，设法保证 $X$ 是红色的。当到达一个新的结点时，要确保 $P$ 是红色的（归纳地，按照我们试图保持的这种不变性）并且 $X$ 和 $T$ 是黑色的（因为不能有两个相连的红色结点）。存在两种主要的情形。

531

首先，设 $X$ 有两个黑色的儿子。此时有三种子情况，它们如图12-17所示。如果 $T$ 也有两个黑色的儿子，那么可以翻转 $X$ 、 $T$ 和 $P$ 的颜色来保持这种不变性。否则， $T$ 的儿子之一是红色的。根据这个儿子结点是哪一个<sup>1</sup>，可以应用图12-17所示的第二种和第三种情形表示的旋转。特别要注意，这种情形对于树叶将是适用的，因为nullNode被认为是黑色的。

其次， $X$ 的儿子之一是红色的。在这种情形下，我们进入下一阶段，得到新的 $X$ 、 $T$ 和 $P$ 。如

1. 如果两个儿子都是红色的，那么可以应用两种旋转中的任一种。通常，在 $X$ 是一个右儿子没有表示出来的情形存在对称的旋转。

果幸运,  $X$ 将落在红色的儿子上, 则可以继续向前进行。如果不是这样, 那么我们知道 $T$ 将是红色的, 而 $X$ 和 $P$ 将是黑色的。可以旋转 $T$ 和 $P$ , 使得 $X$ 的新父亲是红色的; 当然 $X$ 和它的祖父将是黑色的。此时可以回到第一种主情况。

```

1  /**
2   * Internal routine that is called during an insertion if a node has two red
3   * children. Performs flip and rotations. item is the item being inserted.
4   */
5  void handleReorient( const Comparable & item )
6  {
7      // Do the color flip
8      current->color = RED;
9      current->left->color = BLACK;
10     current->right->color = BLACK;
11
12     if( parent->color == RED ) // Have to rotate
13     {
14         grand->color = RED;
15         if( item < grand->element != item < parent->element )
16             parent = rotate( item, grand ); // Start dbl rotate
17         current = rotate( item, great );
18         current->color = BLACK;
19     }
20     header->right->color = BLACK; // Make root black
21 }
22
23 void insert( const Comparable & x )
24 {
25     current = parent = grand = header;
26     nullNode->element = x;
27
28     while( current->element != x )
29     {
30         great = grand; grand = parent; parent = current;
31         current = x < current->element ? current->left : current->right;
32
33         // Check if two red children; fix if so
34         if( current->left->color == RED && current->right->color == RED )
35             handleReorient( x );
36     }
37
38     // Insertion fails if already present
39     if( current != nullNode )
40         return;
41     current = new RedBlackNode( x, nullNode, nullNode );
42
43     // Attach to parent
44     if( x < parent->element )
45         parent->left = current;
46     else
47         parent->right = current;
48     handleReorient( x );
49 }

```

图12-16 插入过程

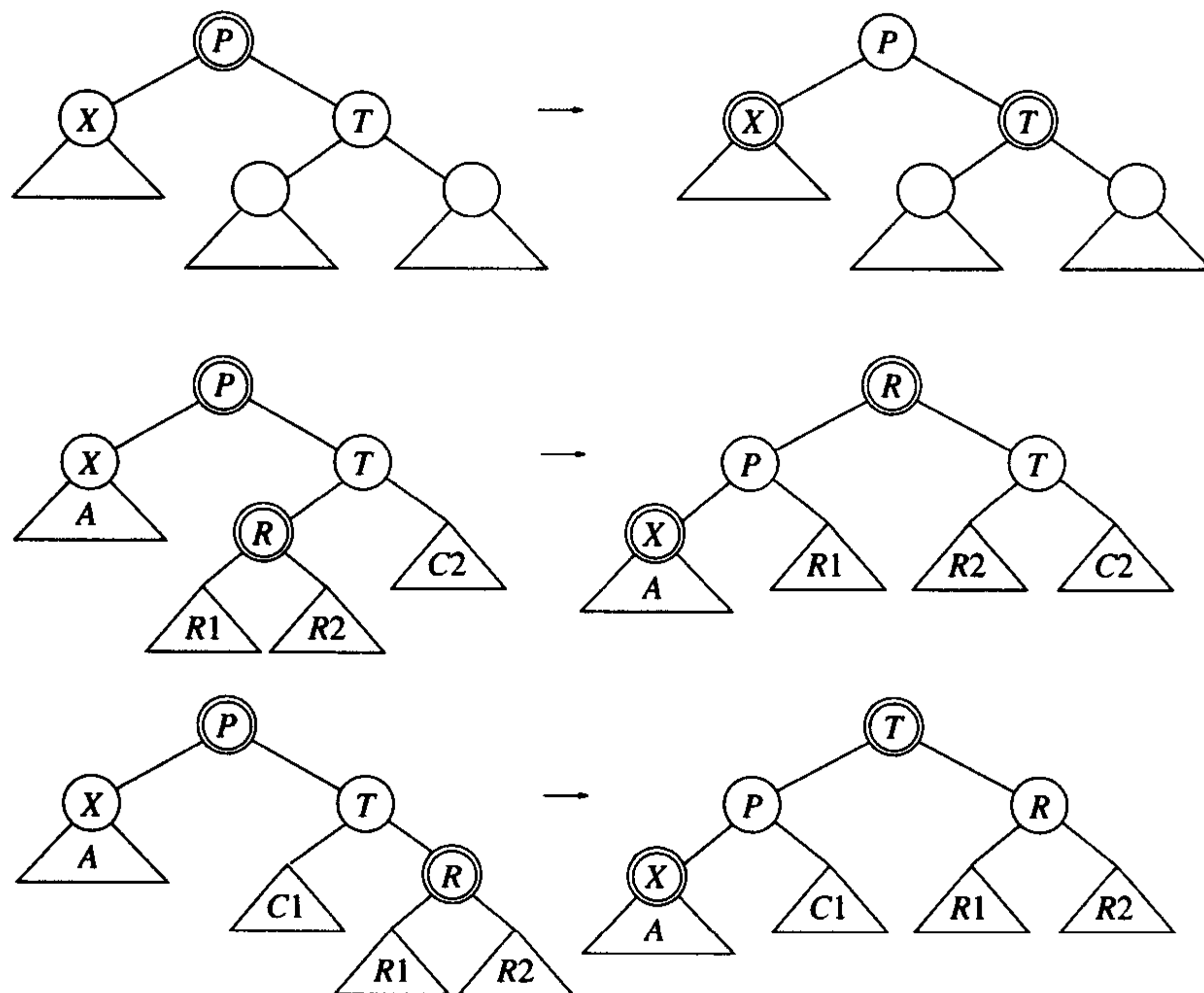


图12-17 当X是一个左儿子并有两个黑色的儿子时的三种情形

## 12.3 确定性跳跃表

用于红黑树的一些想法可以应用到跳跃表以保证对数最坏情形操作。本节描述产生数据结构的最简单的实现方法，即1-2-3确定性跳跃表（deterministic skip list）。

第10章曾介绍过，跳跃表中的结点随机地指定了高度。高度为 $h$ 的结点包含 $h$ 个前向链 $p_1, p_2, \dots, p_h$ ； $p_i$ 链接到高度为 $i$ 或更大的下一个结点。一个结点具有高度 $h$ 的概率为 $0.5^h$ （为了实现时/空交换，0.5可以用0和1.0之间的任何数来代替）。因此，我们期望只处理一些前向链直到下降一层；由于有大约 $\log N$ 层，因此得到每次操作花费 $O(\log N)$ 的期望运行时间。

为使这个界成为最坏情形的界，需要保证只有常数个前向链需要考察直到下降到更低的一层。为此，添加一个平衡条件。首先需要两个定义。

**定义** 两个元素称为是链接的（linked），如果至少存在一个链从一个元素指向另一个元素。

**定义** 两个在高度 $h$ 链接的元素间的间隙容量（gap size）等于它们之间高度为 $h-1$ 的元素的个数。

1-2-3确定性跳跃表满足这样的性质：每一个间隙（除去头和尾之间可能的零间隙外）的容量为1、2或3。作为例子，图12-18显示了一个1-2-3确定性跳跃表。该表有两个容量为3的间隙：第一个是在25和45之间有三个高度为1的元素，第二个是在表头和尾之间有三个高度为2的元素。尾结点包含 $\infty$ ；它的出现简化了算法并使得定义表终端间隙的概念更容易。

显然，沿任一层行进仅仅通过常数个链就可下降到较低的一层。因此，在最坏情形下查找的时间是 $O(\log N)$ 。



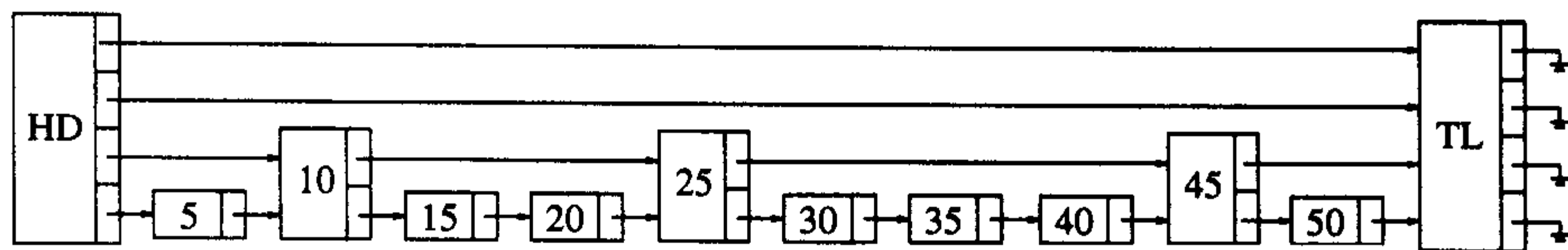


图12-18 一个1-2-3确定性跳跃表

为了执行插入，必须保证当一个高度为 $h$ 的新结点加入进来时不会产生具有4个高度为 $h$ 的结点的间隙。实际上这很简单，采用类似于红黑树中所使用的自顶向下的方法即可。

设现在位于第 $L$ 层上，要降到下一层去。如果要降到的间隙容量是3，那么提升该间隙的中项使其高度为 $L$ ，从而形成两个容量为1的间隙。由于这使得朝向插入的道路上消除了容量为3的间隙，因此插入是安全的。

535

例如，图12-19显示了将项27插入到图12-18的确定性跳跃表中的操作。在头结点上，我们将从第3层降到第2层。由于下降将落入到容量为3的间隙，因此这里的中项（25）将提升到高度3并在表中被拼接好。在第2层的查找将我们带到25，需要在此处下降到第1层。在这里又见到容量为3的间隙，因此把35提升到高度2。结果如图12-20所示。当插入27的时候，将它拼接到表中，如图12-21所示。

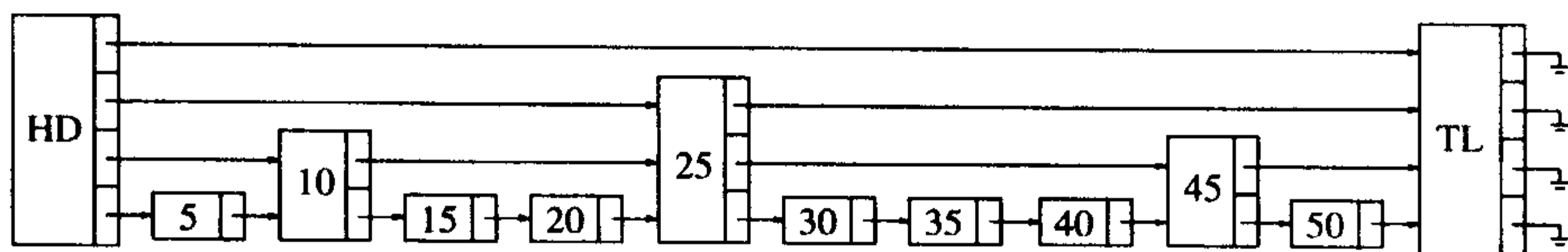


图12-19 插入27：首先，通过提升25将含3个高度2的结点的间隙分裂

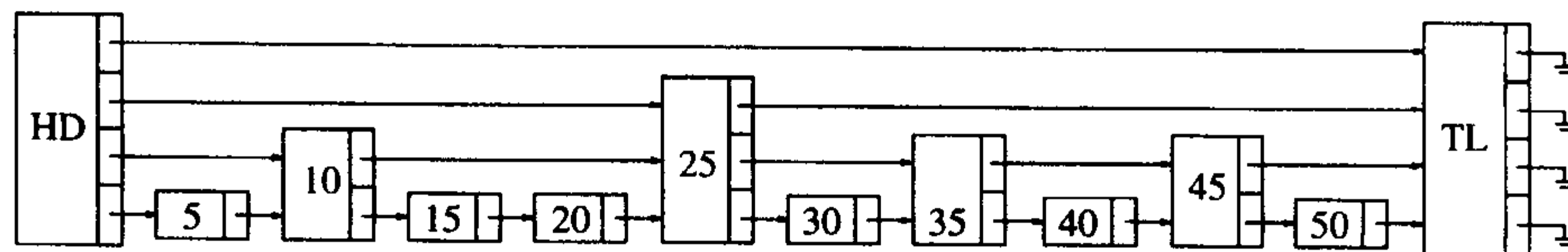


图12-20 插入27：其次，通过提升35将含3个高度1的结点的间隙分裂

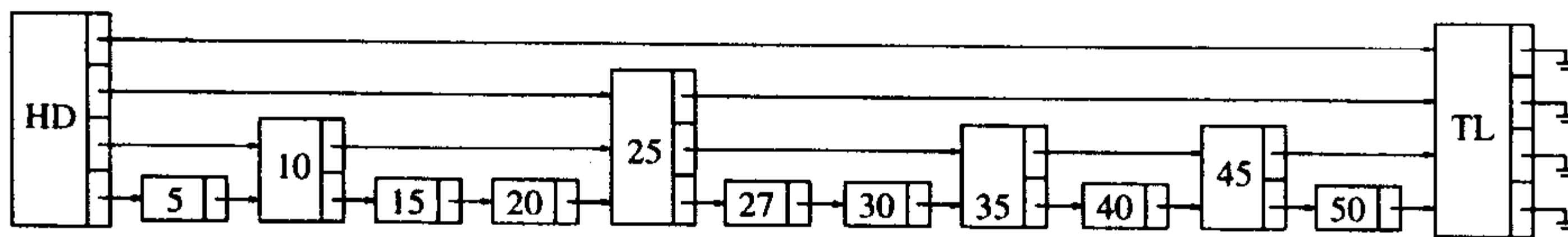


图12-21 插入27：最后，将27作为高度1的结点插入

删除的困难出现在间隙容量为1的情况。当我们看到将要下降到一个容量为1的间隙时，把这个间隙放大：或者通过从相邻间隙（如果容量不为1）借来的方式，或者通过将该间隙与相邻间隙分开的结点的高度降低的方式。由于这两个都是容量为1的间隙，因此结果变成容量为3的间隙。由于有几种情形要处理，因此程序比我们的描述稍微复杂一些。

整个过程是如何实现的呢？在描述了所有的细节之后，我们将看到程序代码的量实际上是相当小的。

第一个重要的细节是，当将一个高 $h$ 的结点提升到高 $h+1$ 的时候，不能花费时间 $O(h)$ 用于将 $h$ 个链复制到一个新数组。否则，插入的时间界就要成为 $O(\log^2 N)$ 了。一种合理的方法是用一个链

表表示高度为 $h$ 的结点中的 $h$ 个前向链。由于是沿着各层向下行进，因此一个结点的链表是以第 $h$ 层前向链开始并以第1层前向链结束。

第二是优化更复杂而且可能占用一些空间。我们不是把结点作为项和前向链的链表来存储，而是存储前向链和前向项对的链表。理解其含义的最容易的方法是参考图12-22，它是图12-21的另一种表示方法。我们将使用术语抽象(abstract)或逻辑(logical)表示来描述图12-21并把图12-22看成是(实际的)实现方法。

536

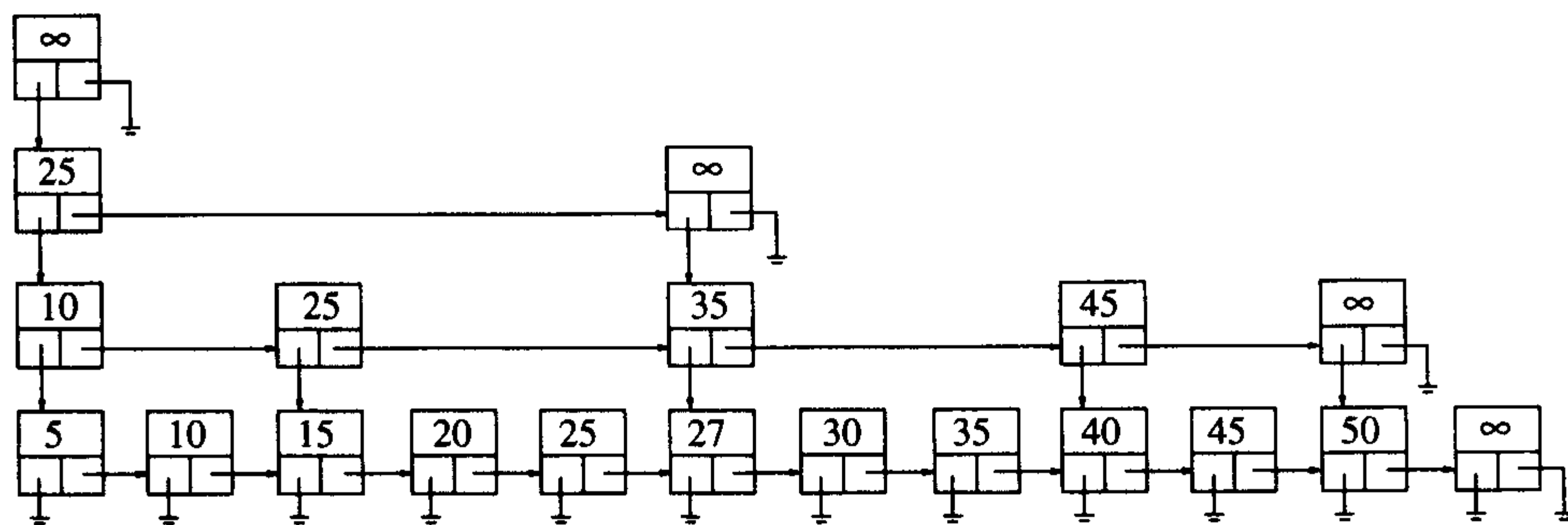


图12-22 图12-21中1-2-3确定性跳跃表的链表实现

首先注意，除了尾结点被去掉外，抽象表示和实际实现二者的“地平线”（即从左到右扫描的高度）是一样的。在我们的实现中，每一个结点都留有使我们下降一层的链、指向同层上的下一个结点的链以及逻辑上存储在下一项中的项（如原始抽象描述所述）。

注意，有些项出现多次，例如，25出现在三个地方。事实上，如果一个结点在抽象表示中的高度为 $h$ ，那么它的项在实际实现中就会出现在 $h$ 个地方。有一些重要的结论和惊人的结果我们将在给出实现方法后进行解释。

基本结点由一个项和两个链组成。为了使编程更快、更简单，这里使用了尾结点；如果不能或不希望赋值 $\infty$ ，那么就必须用别的技巧。我们对头结点和底层结点都有一个标记以代替NULL链。SkipNode类和DSL数据成员如图12-23所示。

查找函数与随机化跳跃表的相同。图12-24指出，如果没有匹配的项，那么或者向下进行，或者向右进行，这取决于比较的结果。如图12-25所示，插入操作由于标记的引入而大大地得到简化。利用某些繁琐的链跟踪可以看到，如果不得不对每一个链是否是NULL进行测试，那么程序代码的规模很容易就扩大三倍。

图12-25指出，确定性跳跃表插入过程的程序多多少少短一些，考虑的情况比红黑树少得多。我们所付出的代价似乎是空间：在最坏情形下有 $2N$ 个结点，每个结点包含两个链和一项。对于红黑树，有 $N$ 个结点，每个结点包含两个链、一项以及一个颜色位（color bit）。因此，可能要用到两倍多的空间。可是，事情没有糟到这一步。首先，经验表明，确定性跳跃表平均使用大约 $1.57N$ 个结点。其次，在某些情况下，确定性跳跃表实际使用的空间少于红黑树。

这里有一个适用于C或C++的实际例子。在32位机上，指针和整数是4个字节。对于某些系统，包括某些版本的UNIX，内存是按块（chunk）来配置的，它们通常是2的幂，但存储管理程序使用4个字节的块。于是，对于12个字节的请求将得到一个16字节块：12个字节由用户使用而4个字节作为系统开销。但是，对于13个字节的需求则必须提供一个32字节块。因此，在这种情况下，确定性跳跃表每个结点使用16个字节，而平均有 $1.57N$ 个结点，故总数一般约为 $25N$ 个字节。可是，红黑树却使用 $32N$ 个字节！这说明在某些机器上附加位是非常昂贵的；这是自组织结构

537

的吸引力之一。

```

1  template <typename Comparable>
2  class DSL
3  {
4      public:
5
6          /**
7           * Construct the tree.
8           * inf is the largest Comparable.
9           */
10         explicit DSL( const Comparable & inf ) : INFINITY( inf )
11         {
12             bottom = new SkipNode( );
13             bottom->right = bottom->down = bottom;
14             tail = new SkipNode( INFINITY );
15             tail->right = tail;
16             header = new SkipNode( INFINITY, tail, bottom );
17         }
18
19         // Additional public member functions (not shown)
20
21     private:
22         struct SkipNode
23         {
24             Comparable element;
25             SkipNode *right;
26             SkipNode *down;
27
28             SkipNode( const Comparable & theElement = Comparable( ),
29                     SkipNode *rt = NULL, SkipNode *dt = NULL )
30                 : element( theElement ), right( rt ), down( dt ) { }
31
32         };
33
34         Comparable INFINITY;
35         SkipNode *header; // The list
36         SkipNode *bottom;
37         SkipNode *tail;
38
39         // Additional private member functions (not shown)
40 };

```

图12-23 确定性跳跃表: SkipNode类和DSL数据成员

```

1  bool contains( const Comparable & x ) const
2  {
3      SkipNode *current = header;
4
5      bottom->element = x;
6      for( ; ; )
7          if( x < current->element )
8              current = current->down;
9          else if( current->element < x )
10             current = current->right;
11          else
12             return current != bottom;
13 }

```

图12-24 确定性跳跃表: contains例程

```

1 void insert( const Comparable & x )
2 {
3     SkipNode *current = header;
4
5     bottom->element = x;
6     while( current != bottom )
7     {
8         while( current->element < x )
9             current = current->right;
10
11         // If gap size is 3 or at bottom level and
12         // must insert, then promote middle element
13         if( current->down->right->right->element < current->element )
14         {
15             current->right = new SkipNode( current->element,
16   current->right, current->down->right->right );
17             current->element = current->down->right->element;
18         }
19         else
20             current = current->down;
21     }
22
23     // Raise height of DSL if necessary
24     if( header->right != tail )
25         header = new SkipNode( INFINITY, tail, header );
26 }

```

图12-25 确定性跳跃表：插入过程

确定性跳跃表的性能似乎比红黑树要强。当寻找插入时间的改进时，代码行

```
if(current->down->right->right->element < current->element)
```

是很好的<sup>1</sup>；如果把一些项存储在最多三个元素的一个数组中，那么对于第三项的访问可以直接进行，而不用再通过两个right指针。图12-26表示的是所得到的结构，具有讽刺意味的是，这个结构很像第4章讨论的B树，我们称之为1-2-3确定性跳跃表的水平数组实现（horizontal array implementation）。正如存在链表形式和水平数组形式的高阶B树一样，也有这两种形式的高阶确定性跳跃表。哪种方法最好还有待研究，而且可能取决于特定的系统和应用。

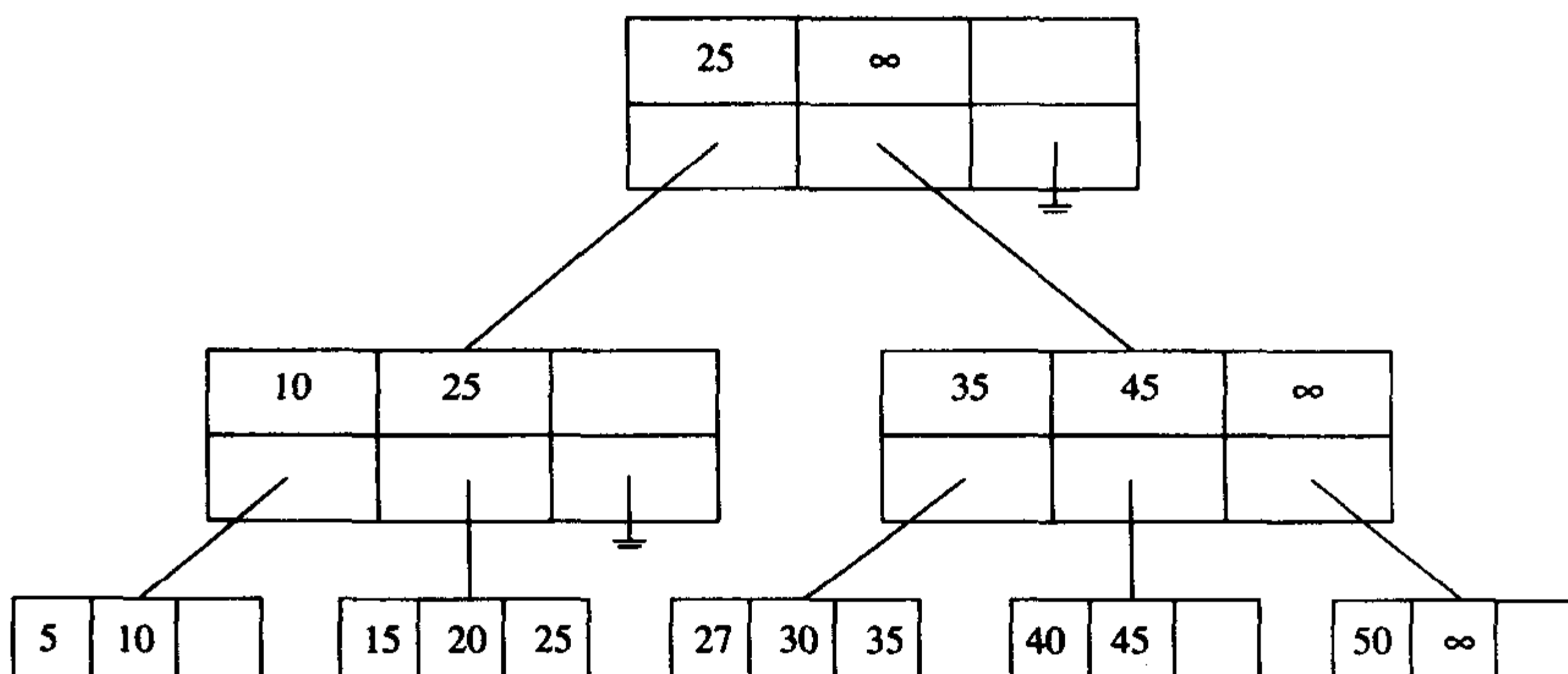


图12-26 图12-22的水平数组实现

1. 事实上，更“明显”的测试 `if(current.down.right.right.right.element.compareTo(current.element) == 0)` 对某些系统多花费 20% 的时间！



## 12.4 AA树

538  
540

由于大量可能的旋转，红黑树的编程相当复杂，特别是删除操作。确定性跳跃表的代码虽在一定程度上要少一些，但仍然是相当复杂的，这由所需的三个标记可以看出。当然，确定性跳跃表中的删除工作也相当复杂。本节描述**二叉B树**（binary B-tree）的一种简单但却颇具竞争力的实现方法，这种树叫作**BB树**。BB树是带有一个附加条件的红黑树：一个结点最多有一个红色的儿子。为使编程容易，我们采纳如下一些法则。

(1) 首先，加入只有右儿子可以是红色的条件，这就消除了约一半的可能重新构建的情形。它也消除了删除算法中一个恼人的情形：如果一个内部结点只有一个儿子，那么这个儿子一定是右儿子（它刚好是红色的），因为黑色左儿子将会违反红黑树的着色性质4。因此，总可以用一个内部结点的右子树中的最小结点代替该内部结点。

(2) 递归地编写这些过程。

(3) 把信息存在一个小的整数（例如8个比特）中，而不是把颜色位（bit）和每个结点一起存储。这个信息就是结点的层次（level）。结点的层次

- 是1，若该结点是树叶。
- 是它的父结点的层次，若该结点是红色的。
- 比它的父结点的层次少1，若该结点是黑色的。

如此得到的结果是一棵AA树。图12-27显示了用于AA树的类型声明。我们再一次使用标记来代表NULL。

541

```

1 public:
2   AATree( )
3   {
4       nullNode = new AANode;
5       nullNode->left = nullNode->right = nullNode;
6       nullNode->level = 0;
7       root = nullNode;
8   }
9
10 private:
11   struct AANode
12   {
13       Comparable element;
14       AANode *left;
15       AANode *right;
16       int level;
17
18       AANode( ) : left( NULL ), right( NULL ), level( 1 ) { }
19       AANode( const Comparable & e, AANode *lt, AANode *rt, int lv = 1 )
20           : element( e ), left( lt ), right( rt ), level( lv ) { }
21   };

```

图12-27 AA树：结点类和AATree初始化

如果将AA结构需求从颜色转换成层次，那么可以看到，左儿子必然比它的父结点恰好低一个层次，而右儿子可能比父结点低0或1个层次（但不会再多）。

**水平链接**（horizontal link）是一个结点与同层次上的儿子之间的连接；这种结构需求使得水平链接是右链接，并且不可能有两个连续的水平链接。图12-28显示了一棵AA树的示例。查找使用通常的算法完成。一个新项的插入总是在底层进行。不过，有两个问题产生：2的插入将产生

一个左水平链接，而45的插入将产生两个连续的右水平链接。

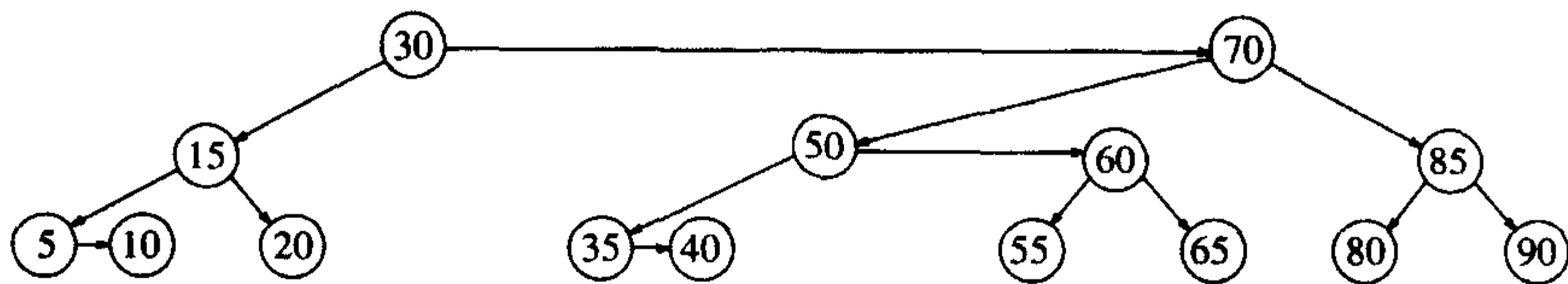


图12-28 插入10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55, 35后得到的一棵AA树

在这两种情况下，一次单旋转都可以使问题得到解决：通过一些右旋转消除左水平链接，而通过一次左旋转消除连续的右水平链接。这些过程分别称为skew和split，如图12-29所示。一次skew除去一个左水平链接，但可能会创建连续的右水平链接；因此我们首先执行skew，然后再执行split。在一次split之后，中间结点R的层次增加。由于新建了一个左水平结点或连续的右水平结点，因而引起X的原始父结点的一些问题，这两个问题都可以通过上述skew/split的方法解决。如果使用递归算法，那么这可以自动地完成。图12-30描述了这两个方法。

```

1 /**
2  * Skew primitive for AA-trees.
3  * t is the node that roots the tree.
4  */
5 void skew( AANode * & t )
6 {
7     if( t->left->level == t->level )
8         rotateWithLeftChild( t );
9 }
10
11 /**
12  * Split primitive for AA-trees.
13  * t is the node that roots the tree.
14  */
15 void split( AANode * & t )
16 {
17     if( t->right->right->level == t->level )
18     {
19         rotateWithRightChild( t );
20         t->level++;
21     }
22 }

```

图12-29 AA树：skew方法和split方法

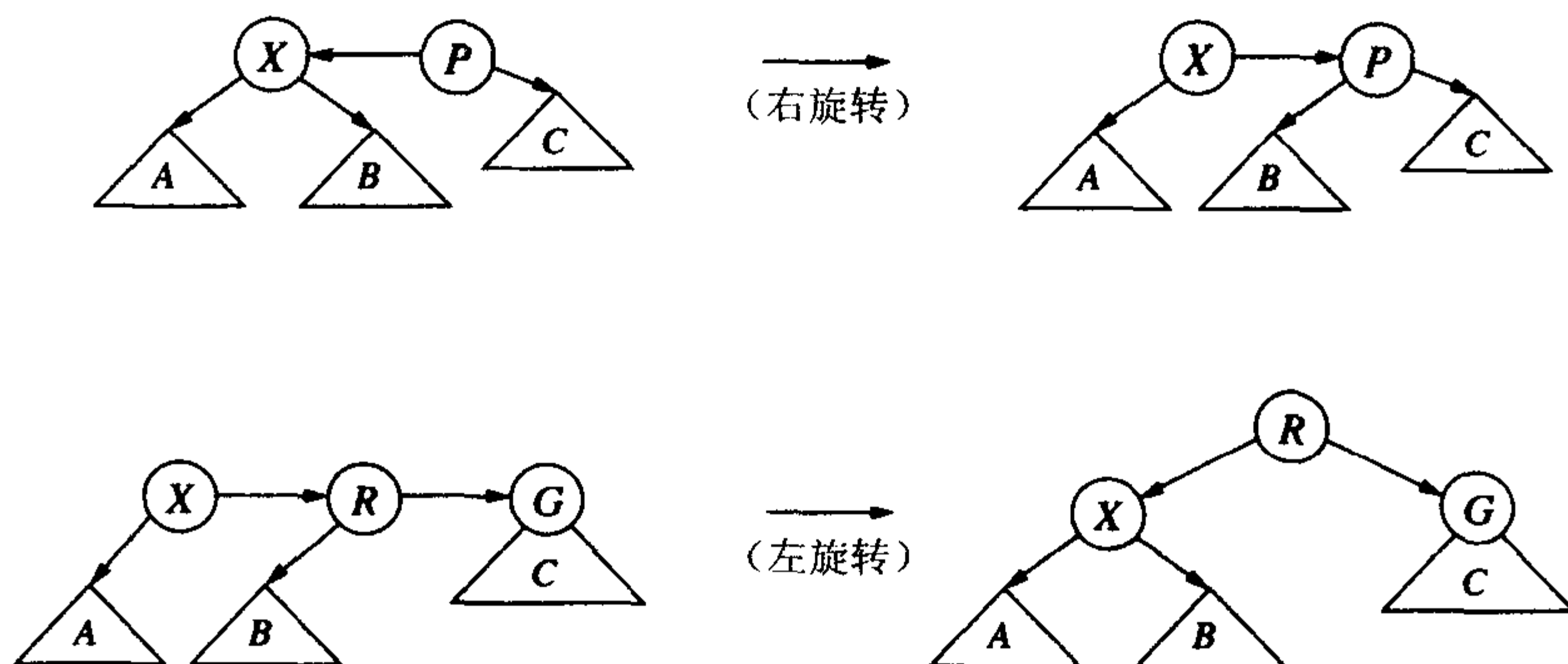


图12-30 skew和split。注意R的层次在一次split中增加

将45插入到图12-28中的AA树的过程如图12-31至图12-35所示。此时的插入过程只比非平衡实现方法多两行，如图12-36所示。

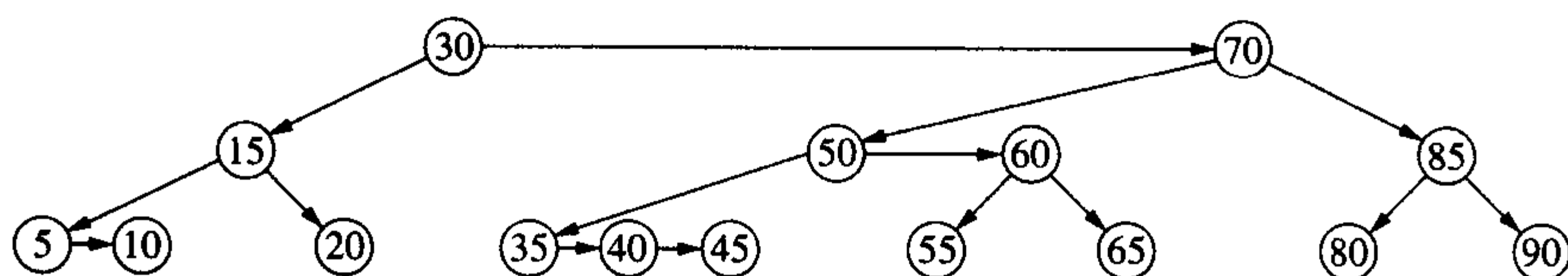


图12-31 在将45插入到示例树中以后

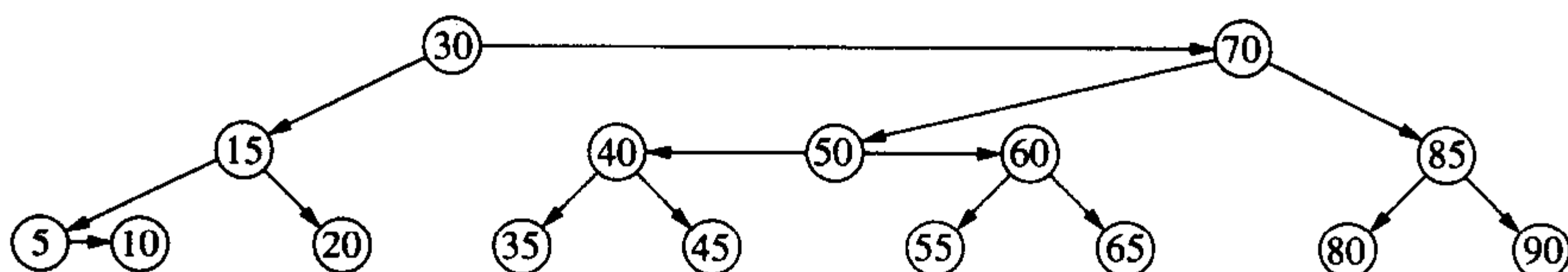


图12-32 在35处进行split之后

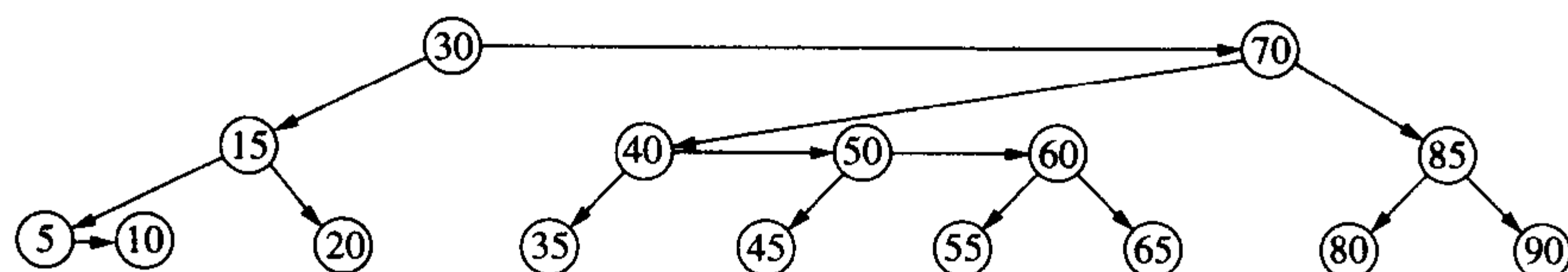


图12-33 在50处进行skew之后

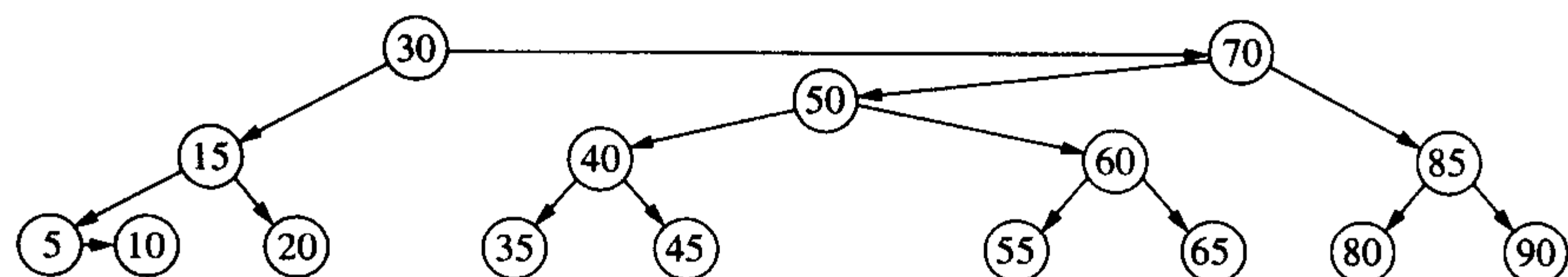


图12-34 在40处进行split之后

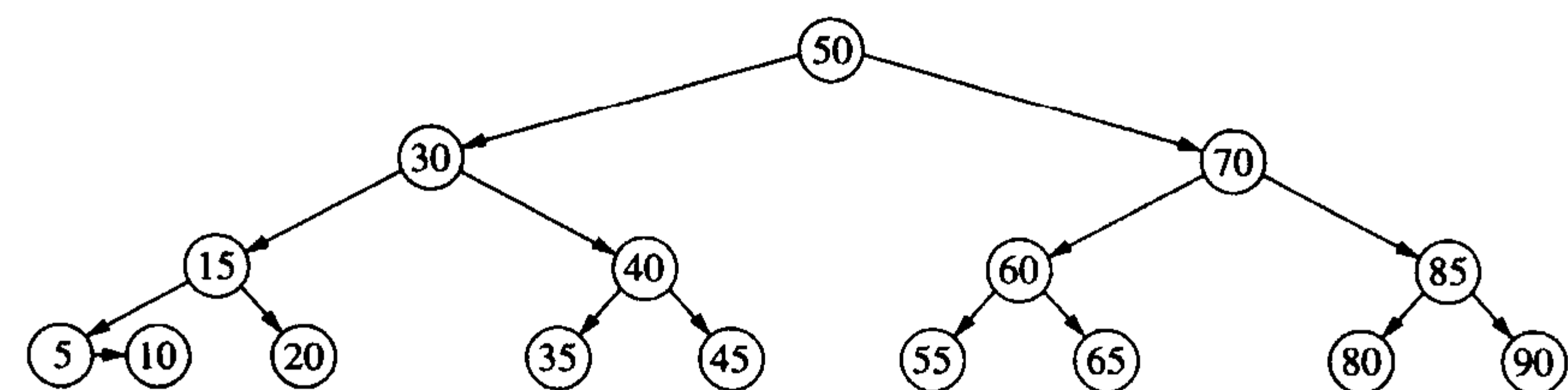


图12-35 在70处进行skew和在30处进行split后得到的树

当然，删除操作更复杂，不过，由于我们除去了许多的特殊情况，程序代码实际上是相当合理的。首先我们知道，如果一个结点不是树叶，那么它必然有一个右儿子，这意味着，当删除一个结点的时候，总可以用其右子树上最小的儿子代替这个结点，这保证它是在第1层上。

为了有助于解决问题，我们使用了两个类变量deletedNode和lastNode。因为remove是递归方法，所以这两个变量都是数据成员。当遍历一个右链接时，调整deletedNode；因为我们递归地调用remove直到到达底部为止（在沿树下行的过程中我们不对相等性进行测试），这保证如

果要删除的项在树上，那么deletedNode将指向包含它的结点<sup>1</sup>。lastNode引用查找终止处的树叶。因为只有到达底部才停止，所以如果该项在树上，那么lastNode将引用层次为1的包含替换值的结点，且必然从该树删除。

```

1  /**
2   * Internal method to insert into a subtree.
3   * x is the item to insert.
4   * t is the node that roots the tree.
5   * Set the new root of the subtree.
6   */
7  void insert( const Comparable & x, AANode * & t )
8  {
9      if( t == nullNode )
10         t = new AANode( x, nullNode, nullNode );
11     else if( x < t->element )
12         insert( x, t->left );
13     else if( t->element < x )
14         insert( x, t->right );
15     else
16         return; // Duplicate; do nothing
17
18     skew( t );
19     split( t );
20 }

```

图12-36 AA树：插入方法

当到达树的底部时，我们执行第2步，将第1层结点值复制到内部结点上然后绕过这个第1层结点。

为了查看非叶结点的层次是否被一次递归调用所破坏，需要检查这些非叶结点。令 $T$ 为当前结点。如果删除将 $T$ 的一个儿子（实际上只有一个由递归调用所输入的儿子可能受影响，但为简单起见我们不跟踪它）的层次降低到比 $T$ 的层次低2，那么 $T$ 的层次也需要降低。此外，如果 $T$ 有一个红色的右儿子，那么 $T$ 的右儿子也必须将它的层次降低。此时，可能在同一层次上有6个结点： $T$ 、 $T$ 的红色的右儿子 $R$ 、 $R$ 的两个儿子以及这些儿子的红色的右儿子。图12-37给出了最简单的可能情况。

544

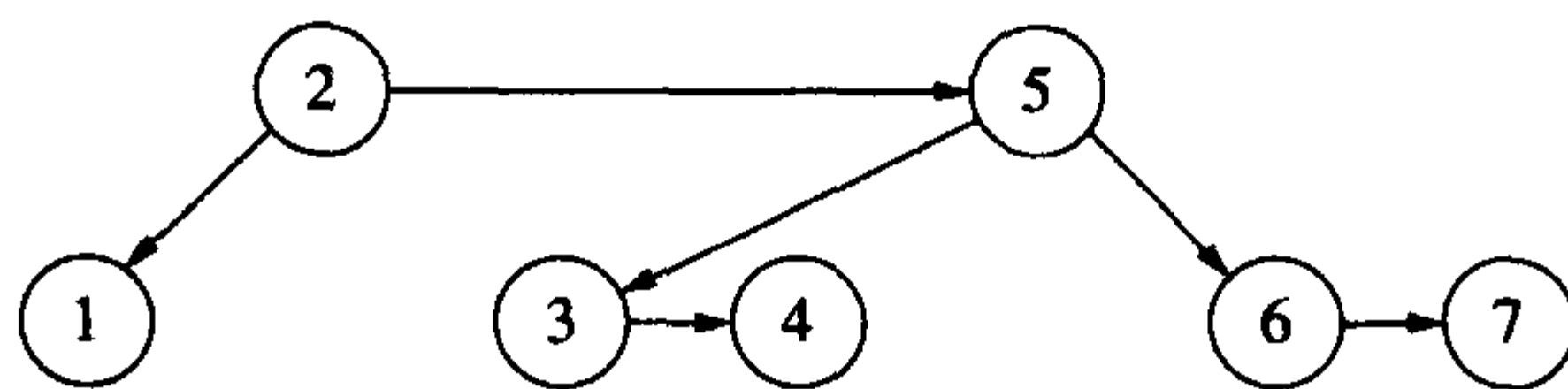


图12-37 当1被删除时，引入水平左链接，所有结点的层次变成1。使所有的链接指向右侧是通过三次调用skew来完成的。通过两次调用split移去水平链接。

在结点1删除以后，结点2和结点5变成了第1层的结点。首先，必须调整在结点5和3之间引入的左水平链接，这基本上需要两次旋转（一次是在结点5和3之间，而后是在结点5和4之间）。在这种情况下不涉及当前结点 $T$ 。另一方面，如果删除来自右边，那么 $T$ 的左结点可能突然之间变成水平的了；这也需要一次类似的双旋转（在 $T$ 处开始）。为了避免测试所有这些情形，我们调用了三次skew。一旦调用完成，则再调用两次split就足以重新安排这些水平的边。整个删除例程如

1. 这个技巧可以用于 contains 方法，用每个结点的两路比较代替在每个结点所做的三路比较，外加在底部进行的相等性测试。



545  
546

图12-38所示。从各方面来看，这对编程来说都是相对简单的数据结构。

```

1  /**
2   * Internal method to remove from a subtree.
3   * x is the item to remove.
4   * t is the node that roots the tree.
5   * Set the new root of the subtree.
6   */
7  void remove( const Comparable & x, AANode * & t )
8  {
9      static AANode *lastNode, *deletedNode = nullNode;
10
11     if( t != nullNode )
12     {
13         // Step 1: Search down the tree and set lastNode and deletedNode
14         lastNode = t;
15         if( x < t->element )
16             remove( x, t->left );
17         else
18         {
19             deletedNode = t;
20             remove( x, t->right );
21         }
22
23         // Step 2: If at the bottom of the tree and
24         //          x is present, we remove it
25         if( t == lastNode )
26         {
27             if( deletedNode == nullNode || x != deletedNode->element )
28                 return; // Item not found; do nothing
29             deletedNode->element = t->element;
30             deletedNode = nullNode;
31             t = t->right;
32             delete lastNode;
33         }
34         // Step 3: Otherwise, we are not at the bottom; rebalance
35         else
36             if( t->left->level < t->level - 1 ||
37                 t->right->level < t->level - 1 )
38             {
39                 if( t->right->level > --t->level )
40                     t->right->level = t->level;
41                 skew( t );
42                 skew( t->right );
43                 skew( t->right->right );
44                 split( t );
45                 split( t->right );
46             }
47     }
48 }

```

图12-38 AA树：删除过程

## 12.5 treap树

最后一种二叉查找树可能是最简单的一种，叫作**treap**树。它像跳跃表一样，使用随机数并且对任意的输入都能给出 $O(\log N)$ 的期望时间的性能。查找时间等同于非平衡二叉查找树（从而比平衡查找树要慢），而插入时间只比递归非平衡二叉查找树的实现方法稍慢。虽然删除操作要

慢得多，但仍然是 $O(\log N)$ 期望时间。

treap树非常简单，不用画图就可描述它。树中的每个结点存储一项、一个左和右指针以及一个优先级，优先级是建立结点时随机指定的。treap树就是二叉查找树，但其结点优先级满足堆序的性质：任意结点的优先级必须至少和它父亲的优先级一样大。

每一项都有不同优先级的不同项的集合只能由一棵treap树表示。这很容易由归纳法推导，因为具有最低优先级的结点必然是根。因此，树是根据优先级的 $N!$ 种可能的排列而不是根据项的 $N!$ 种排序形成的。结点的声明很简单，只要求priority数据成员的增加，如图12-39所示。标记nullNode的优先级为 $\infty$ 。

```

1  template <typename Comparable>
2  class Treap
3  {
4      public:
5          Treap( )
6          {
7              nullNode = new TreapNode;
8              nullNode->left = nullNode->right = nullNode;
9              nullNode->priority = INT_MAX;
10             root = nullNode;
11         }
12
13     Treap( const Treap & rhs );
14     ~Treap( );
15     // Additional public member functions (not shown)
16
17     private:
18     struct TreapNode
19     {
20         Comparable element;
21         TreapNode *left;
22         TreapNode *right;
23         int priority;
24
25         TreapNode( ) : left( NULL ), right( NULL ), priority( INT_MAX ) { }
26         TreapNode( const Comparable & e, TreapNode *lt, TreapNode *rt, int pr )
27             : element( e ), left( lt ), right( rt ), priority( pr )
28             { }
29     };
30
31     TreapNode *root;
32     TreapNode *nullNode;
33     Random randomNums;
34     // Additional private member functions (not shown)
35 };

```

图12-39 Treap类接口和构造函数

treap树的插入操作也简单：在一项作为树叶加入之后，将它沿着treap树向上旋转直到其优先级满足堆序为止。可以证明旋转的期望次数小于2。在要被删除的项找到以后，通过把它的优先级增加到 $\infty$ 并沿着低优先级儿子的路径向下旋转可将其删除。一旦它是树叶，就可以把它除去。图12-40和图12-41中的例程利用递归实现这些策略，非递归的实现方法留给读者（练习12.17）。注意，对于删除，当结点逻辑上是树叶时，它仍然有nullNode作为它的左儿子和右儿子。因此，它与右儿子旋转，旋转后，t为nullNode，而存储要被删除的项的左儿子可能被释放。还要注意，我们的实现假设没有重复元；如果这个假设不成立，那么remove可能失败（为什么？）。

```

1 /**
2  * Internal method to insert into a subtree.
3  * x is the item to insert.
4  * t is the node that roots the tree.
5  * Set the new root of the subtree.
6  * (randomNums is a Random object that is a data member of Treap.)
7  */
8 void insert( const Comparable & x, TreapNode* & t )
9 {
10     if( t == nullNode )
11         t = new TreapNode( x, nullNode, nullNode, randomNums.randomInt( ) );
12     else if( x < t->element )
13     {
14         insert( x, t->left );
15         if( t->left->priority < t->priority )
16             rotateWithLeftChild( t );
17     }
18     else if( t->element < x )
19     {
20         insert( x, t->right );
21         if( t->right->priority < t->priority )
22             rotateWithRightChild( t );
23     }
24     // else duplicate; do nothing
25 }

```

图12-40 treap树：插入例程

```

1 /**
2  * Internal method to remove from a subtree.
3  * x is the item to remove.
4  * t is the node that roots the tree.
5  * Set the new root of the subtree.
6  */
7 void remove( const Comparable & x, TreapNode * & t )
8 {
9     if( t != nullNode )
10     {
11         if( x < t->element )
12             remove( x, t->left );
13         else if( t->element < x )
14             remove( x, t->right );
15         else
16         {
17             // Match found
18             if( t->left->priority < t->right->priority )
19                 rotateWithLeftChild( t );
20             else
21                 rotateWithRightChild( t );
22
23             if( t != nullNode ) // Continue on down
24                 remove( x, t );
25             else
26             {
27                 delete t->left;
28                 t->left = nullNode; // At a leaf
29             }
30         }
31     }
32 }

```

图12-41 treap树：删除过程

treap树之所以特别容易实现，是因为我们绝对不必担心调整priority数据成员。平衡树处理方法的困难之一是追查由于未能更新操作过程中的信息而导致的错误。从合理的插入和删除程序包的全部程序行来看，treap树，特别是非递归方法的实现，似乎才是不费力的赢家。

548

## 12.6 k-d树

设一家广告公司拥有一个数据库并需要为某些客户生成邮寄标签。典型的要求可能是需要散发邮件给那些年龄在34~49之间且年收入在100 000~150 000美元之间的人。这个问题叫作二维范围查询（two-dimensional range query）。在一维情况下，该问题可以借助于简单的递归算法通过遍历预先构造的二叉查找树以 $O(M + \log N)$ 平均时间解决。这里， $M$ 是由查询所报告的匹配的个数。我们希望对二维或更高维的情况得到类似的界。

二维查找树（two-dimensional search tree）具有简单的性质：在奇数层上的分支按照第一个键进行，而在偶数层上的分支按照第二个键进行。根是任意选取的奇数层。图12-42表示一棵2-d树。2-d树的插入操作是二叉查找树插入操作的平凡的扩展：在沿树下行时，我们需要保留当前的层。为保持程序代码简单，假设基本项是两个元素的数组。此时需要把层限制在0和1之间。图12-43显示的是执行插入的程序。在本节中我们使用递归；实践中使用的非递归实现方法是很简单的，把它留作练习12.23。特别是由于若干项在一个键上可能相同，因此困难之一是重复元问题。我们的程序允许重复元，并且总是把它们放在右分支上；显然，如果有太多的重复元，那么这可能就是一个问题。

549

550

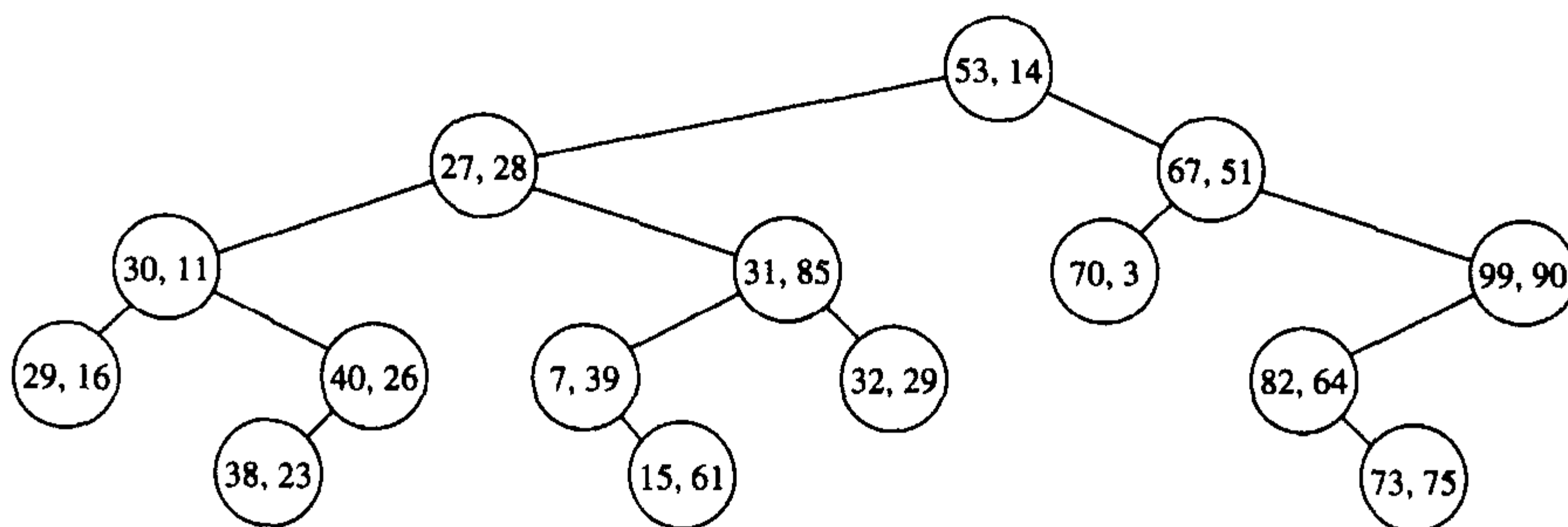


图12-42 2-d树示例

```

1 public:
2   void insert( const vector<Comparable> & x )
3   {
4       insert( x, root, 0 );
5   }
6
7 private:
8   void insert( const vector<Comparable> & x, KdNode * & t, int level )
9   {
10      if( t == NULL )
11          t = new KdNode( x );
12      else if( x[ level ] < t->data[ level ] )
13          insert( x, t->left, 1 - level );
14      else
15          insert( x, t->right, 1 - level );
16  }

```

图12-43 2-d树的插入



稍加思索便可确信，随机构造的2-d树与随机二叉查找树具有相同的结构化性质：高度平均为 $O(\log N)$ ，但最坏情形则是 $O(N)$ 。

不像二叉查找树有精巧的 $O(\log N)$ 最坏情形的变体存在，不存在已知的方案能够保证一棵平衡的2-d树。问题在于，这样一种方案很可能基于树旋转，而树旋转在2-d树中是行不通的。最好的办法是通过重新构造子树来定期地对树进行平衡，具体描述可见练习。类似地，也不存在超越明显的懒惰删除方法的删除算法。如果在需要处理查询之前所有的项都已到达，那么就能够以 $O(M \log N)$ 时间构造一棵理想平衡2-d树；这就是练习12.21(c)。

551

有几种查询可以在2-d树上进行。我们可以要求精确的匹配，或者基于两个键之一的匹配；后者称为**部分匹配查询**（partial match query）。这两种都是（正交）**范围查询**（range query）的特殊情形。

正交范围查询给出其第一个键在一个特定的值集合之间且第二个键在另一个特定的值集合之间的所有项。这正是本节介绍中所描述的问题。如图12-44所示，范围查询通过一次递归的树遍历很容易解出。通过在递归调用之前进行测试，可以避免不必要地访问所有结点。

```

1 public:
2     /**
3      * Print items satisfying
4      * low[ 0 ] <= x[ 0 ] <= high[ 0 ] and
5      * low[ 1 ] <= x[ 1 ] <= high[ 1 ]
6      */
7     void printRange( const vector<Comparable> & low,
8                     const vector<Comparable> & high ) const
9     {
10        printRange( low, high, root, 0 );
11    }
12
13 private:
14     void printRange( const vector<Comparable> & low,
15                     const vector<Comparable> & high,
16                     KdNode *t, int level ) const
17     {
18        if( t != NULL )
19        {
20            if( low[ 0 ] <= t->data[ 0 ] && high[ 0 ] >= t->data[ 0 ] &&
21                low[ 1 ] <= t->data[ 1 ] && high[ 1 ] >= t->data[ 1 ] )
22                cout << "(" << t->data[ 0 ] << ", "
23                    << t->data[ 1 ] << ")" << endl;
24
25            if( low[ level ] <= t->data[ level ] )
26                printRange( low, high, t->left, 1 - level );
27            if( high[ level ] >= t->data[ level ] )
28                printRange( low, high, t->right, 1 - level );
29        }
30    }

```

图12-44 2-d树：范围查找

为找到特定的项，可以令low等于high且等于要查找的项。为了执行一次部分匹配查询，令这次匹配中涉及不到的键的范围为 $-\infty \sim \infty$ 。而其余的范围则设置为使低点和高点等于匹配中所涉及的键的值。

552

在2-d树中插入或精确匹配查找花费的时间平均正比于树的深度，即 $O(\log N)$ ，而在最坏情形下为 $O(N)$ 。范围查找的运行时间依赖于如何对平衡树，是否要求部分匹配，以及实际上有多少项

被找到。我们提出三个已经被证明的结果。

对于理想平衡树，一次范围查询要报告 $M$ 次匹配在最坏情形下可能花费 $O(M + \sqrt{N})$ 时间。在任一结点，可能必须访问4个孙子中的2个，于是方程 $T(N) = 2T(N/4) + O(1)$ 成立。然而在实践中，这些查找趋向于非常有效，甚至最坏情形都不是那么差，因为对于典型的 $N$ ， $\sqrt{N}$ 和 $\log N$ 之间的差由隐藏于大 $O$ 记法中的更小常数补偿。

对于随机构造的树，部分匹配查询的平均运行时间为 $O(M + N^\alpha)$ ，其中 $\alpha = (-3 + \sqrt{17})/2$ （见下）。最近的多少令人诧异的结果是它基本上描述了随机2-d树的一次范围查找的平均运行时间。

对于 $k$ 维的情况，同样的算法仍然成立；我们通过每层上的键进行循环。不过，在实践中平衡开始变得越来越差，因为一般重复元和非随机输入的影响变得更为明显。我们把编程的细节留给读者作为练习而只叙述分析结果：对于理想平衡树，一次范围查询的最坏情形运行时间为 $O(M + kN^{1-1/k})$ 。在随机构造的 $k$ -d树中，涉及 $k$ 个键中的 $p$ 个键的部分匹配查询花费 $O(M + N^\alpha)$ ，其中 $\alpha$ 是方程

$$(2 + \alpha)^p(1 + \alpha)^{k-p} = 2^k$$

（唯一）的正根。对各种 $p$ 和 $k$ ， $\alpha$ 的计算留作练习； $k=2$ 和 $p=1$ 的情况下的值反映在上面对于随机2-d树的部分匹配所叙述的结果中。

虽然有几种新奇的结构支持范围查找，但是 $k$ -d树恐怕是达到可接受的运行时间的最简单结构。

## 12.7 配对堆

我们考察的最后一个数据结构是配对堆（pairing heap）。对配对堆的分析问题仍然未解决，不过，当需要decreaseKey操作的时候，它似乎胜过其他的堆结构。其效率高的最可能的原因是其简单性。配对堆被表示成堆序树。图12-45给出了一个配对堆示例。

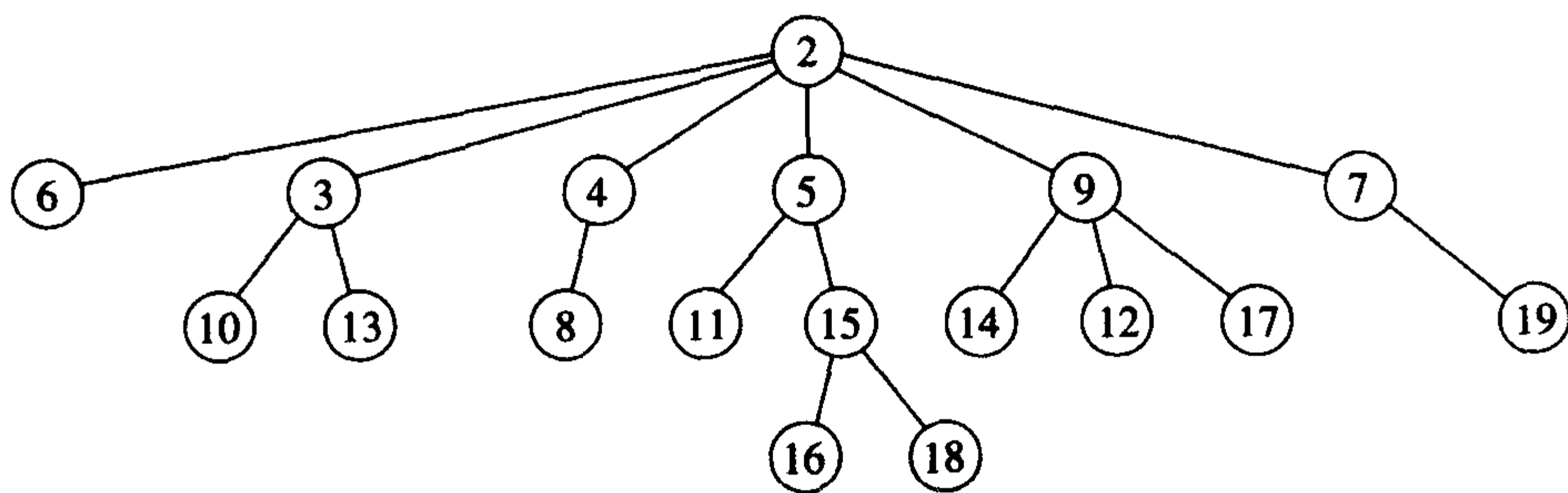


图12-45 示例配对堆：抽象表示

配对堆的具体实现用到第4章中讨论的左儿子、右兄弟表示法。我们将看到，decreaseKey操作要求每个结点包含一个附加的链。作为最左儿子的结点含有一个指向其父亲的链；否则这个结点就是一个右兄弟并且含有一个指向它的左兄弟的链。我们把这个数据成员叫作prev。为了简洁，这里省去类框架和配对堆结点的声明；它们完全是直观的。图12-46指出图12-45中的配对堆的实际表示。

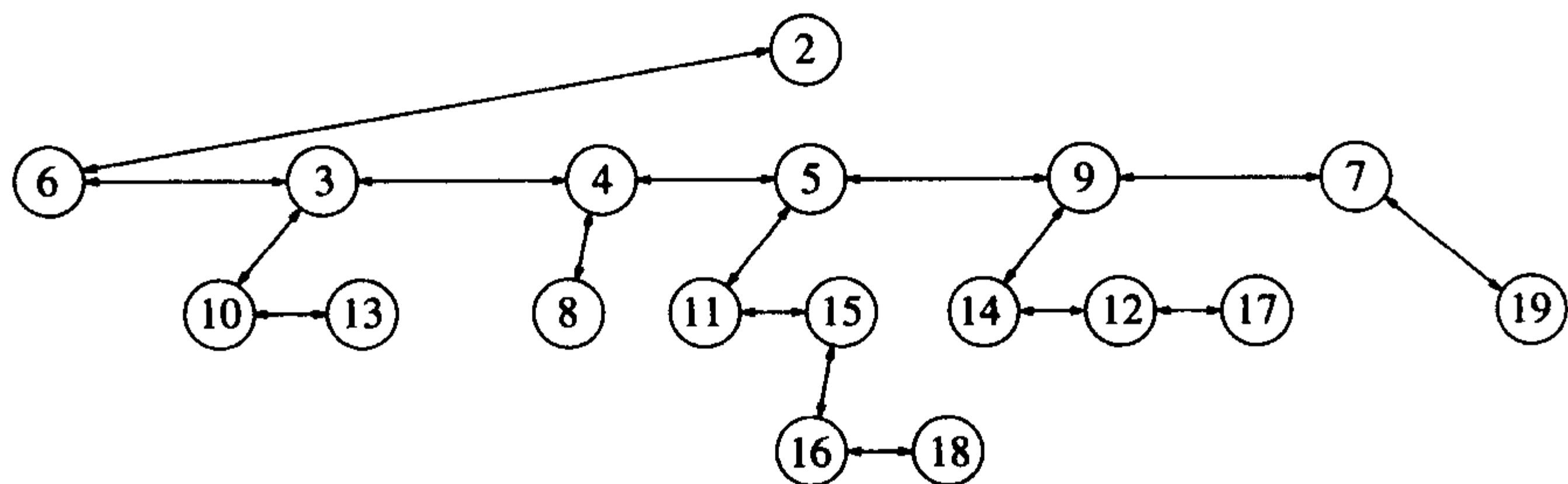


图12-46 前面的配对堆的具体表示

553 现在从概述基本操作开始。为了合并两个配对堆，使具有较大根的堆成为具有较小根的堆的左儿子。当然，插入是合并的特殊情形。为执行一次decreaseKey，降低被请求结点的值。因为对于所有的结点都不保存父指针，所以我们不知道这是否会破坏堆序。如此，将调整后的结点从它的父结点切除，通过合并所得到的两个堆来完成decreaseKey操作。为了执行deleteMin，将根除去，创建堆的一个集合。如果根有 $c$ 个儿子，那么对合并过程进行 $c-1$ 次调用将重建该堆。这里，最重要的细节就是用于执行合并的方法以及如何应用 $c-1$ 次合并。

图12-47显示了如何合并两个子堆。这个过程可被推广到允许第二个子堆有兄弟的情形。前面提到过，可以让具有较大根的子堆成为另一个子堆的最左儿子。程序很简单，如图12-48所示。注意，我们有几个实例，在这些实例中，在给结点的引用赋予prev数据成员之前要测试它是否是NULL；这使我们想到，有一个nullNode标记或许是有用的，它习惯上放在本章的查找树的实现中。

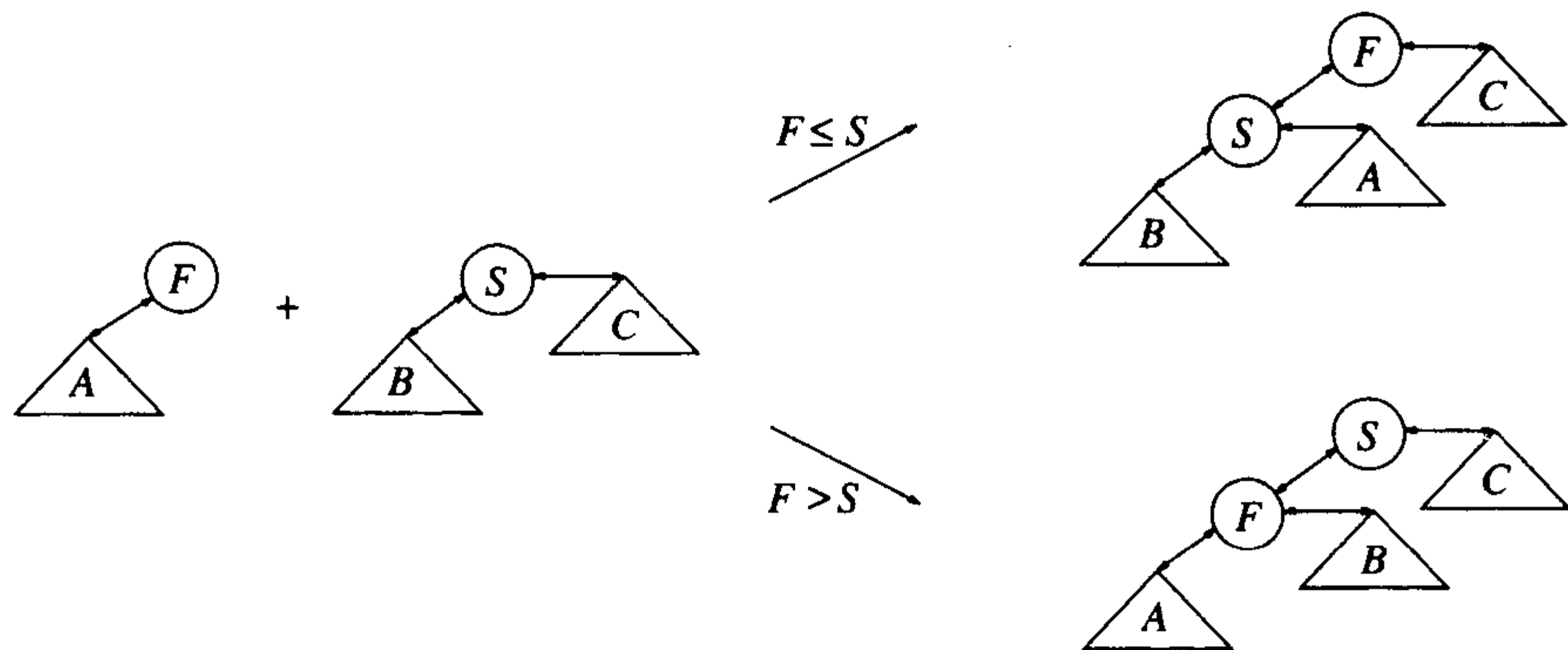


图12-47 compareAndLink合并两个子堆

insert和decreaseKey操作是抽象描述的简单实现。decreaseKey需要一个位置（position）对象，即PairNode\*。由于项在其第一次插入时确定（不可改变），因此insert将指向PairNode的指针返回给调用者。程序如图12-49所示。如果新的键值不小于旧的，那么decreaseKey的例程抛出一个异常；否则，得到的结构可能不符合堆序性质。基本的deleteMin过程由抽象描述直接得到，如图12-50所示。

554 当然，麻烦在于一些细节上：combineSiblings如何实现？已经提出几种变体，但是没有人能证明它们能够提供与斐波那契堆相同的摊还界。最近已经证明，事实上几乎所有提出的方法在理论上都不如斐波那契堆有效。即使这样，对于涉及大量decreaseKey操作的一般图论应用来说，图12-51中的方法似乎总是和其他堆结构一样运行甚至比它们（包括二叉堆）还好。

```

1  /**
2   * Internal method that is the basic operation to maintain order.
3   * Links first and second together to satisfy heap order.
4   * first is root of tree 1, which may not be NULL.
5   * first->nextSibling MUST be NULL on entry.
6   * second is root of tree 2, which may be NULL.
7   * first becomes the result of the tree merge.
8   */
9  void compareAndLink( PairNode * & first, PairNode *second )
10 {
11     if( second == NULL )
12         return;
13
14     if( second->element < first->element )
15     {
16         // Attach first as leftmost child of second
17         second->prev = first->prev;
18         first->prev = second;
19         first->nextSibling = second->leftChild;
20         if( first->nextSibling != NULL )
21             first->nextSibling->prev = first;
22         second->leftChild = first;
23         first = second;
24     }
25     else
26     {
27         // Attach second as leftmost child of first
28         second->prev = first;
29         first->nextSibling = second->nextSibling;
30         if( first->nextSibling != NULL )
31             first->nextSibling->prev = first;
32         second->nextSibling = first->leftChild;
33         if( second->nextSibling != NULL )
34             second->nextSibling->prev = second;
35         first->leftChild = second;
36     }
37 }

```

图12-48 配对堆：合并两个子堆的例程

这是已经提出的许多变体方法中最简单和最实际的方法，称为两趟合并法（two-pass merging）。首先，从左到右扫描，合并诸儿子对<sup>1</sup>。在第一次扫描之后，有一半树要合并。然后执行第二趟扫描，从右到左。在每一步，将第一次扫描剩下的最右边的树和当前合并的结果合并。例如，如果有8个儿子 $c_1 \sim c_8$ ，那么第一次扫描进行 $c_1$ 和 $c_2$ ， $c_3$ 和 $c_4$ ， $c_5$ 和 $c_6$ ， $c_7$ 和 $c_8$ 的合并，得到结果 $d_1, d_2, d_3$ 和 $d_4$ 。再通过合并 $d_3$ 和 $d_4$ 执行第二趟扫描；然后 $d_2$ 和这个结果合并，最后 $d_1$ 再和得到的结果合并。

这里的实现方法要求一个数组存储诸子树。在最坏情形下，可能有 $N-1$ 项都是根的儿子，但是在combineSiblings函数的内部声明一个大小为 $N$ 的（非static）数组将给出一个 $O(N)$ 算法。因此，我们用一个扩大的数组来代替。因为它是static的，在每次调用中都重复使用，从而不需要过多的重复初始化。

另外一些合并方法在练习中讨论。唯一简单的而且容易发现缺欠的合并策略是从左到右单趟合并（练习12.35）。配对堆是“简单即更好”的一个很好的例子，并且似乎是要要求decreaseKey或merge操作的一些重要应用所选择的方法。

1. 如果有奇数个儿子必须小心处理。此时，将最后一个儿子与最右合并的结果合并以完成第一趟扫描。



```

1 struct PairNode;
2 typedef PairNode * Position;
3
4 /**
5  * Insert item x into the priority queue, maintaining heap order.
6  * Return the Position (a pointer to the node) containing the new item.
7  */
8 Position insert( const Comparable & x )
9 {
10     PairNode *newNode = new PairNode( x );
11
12     if( root == NULL )
13         root = newNode;
14     else
15         compareAndLink( root, newNode );
16     return newNode;
17 }
18
19 /**
20  * Change the value of the item stored in the pairing heap.
21  * Throw IllegalArgumentException if newVal is larger than
22  * currently stored value.
23  * p is a Position returned by insert.
24  * newVal is the new value, which must be smaller
25  * than the currently stored value.
26  */
27 void decreaseKey( Position p, const Comparable & newVal )
28 {
29     if( p->element < newVal )
30         throw IllegalArgumentException( ); // newVal cannot be bigger
31     p->element = newVal;
32     if( p != root )
33     {
34         if( p->nextSibling != NULL )
35             p->nextSibling->prev = p->prev;
36         if( p->prev->leftChild == p )
37             p->prev->leftChild = p->nextSibling;
38         else
39             p->prev->nextSibling = p->nextSibling;
40
41         p->nextSibling = NULL;
42         compareAndLink( root, p );
43     }
44 }

```

图12-49 配对堆: insert和decreaseKey

```

1 void deleteMin( )
2 {
3     if( isEmpty( ) )
4         throw UnderflowException( );
5
6     PairNode *oldRoot = root;
7
8     if( root->leftChild == NULL )
9         root = NULL;
10    else
11        root = combineSiblings( root->leftChild );
12
13    delete oldRoot;
14 }

```

图12-50 配对堆deleteMin

```

1  /**
2   * Internal method that implements two-pass merging.
3   * firstSibling the root of the conglomerate and is assumed not NULL.
4   */
5  PairNode * combineSiblings( PairNode *firstSibling )
6  {
7      if( firstSibling->nextSibling == NULL )
8          return firstSibling;
9
10     // Allocate the array
11     static vector<PairNode *> treeArray( 5 );
12
13     // Store the subtrees in an array
14     int numSiblings = 0;
15     for( ; firstSibling != NULL; numSiblings++ )
16     {
17         if( numSiblings == treeArray.size( ) )
18             treeArray.resize( numSiblings * 2 );
19         treeArray[ numSiblings ] = firstSibling;
20         firstSibling->prev->nextSibling = NULL; // break links
21         firstSibling = firstSibling->nextSibling;
22     }
23     if( numSiblings == treeArray.size( ) )
24         treeArray.resize( numSiblings + 1 );
25     treeArray[ numSiblings ] = NULL;
26
27     // Combine subtrees two at a time, going left to right
28     int i = 0;
29     for( ; i + 1 < numSiblings; i += 2 )
30         compareAndLink( treeArray[ i ], treeArray[ i + 1 ] );
31
32     int j = i - 2;
33
34     // j has the result of last compareAndLink.
35     // If an odd number of trees, get the last one.
36     if( j == numSiblings - 3 )
37         compareAndLink( treeArray[ j ], treeArray[ j + 2 ] );
38
39     // Now go right to left, merging last tree with
40     // next to last. The result becomes the new last.
41     for( ; j >= 2; j -= 2 )
42         compareAndLink( treeArray[ j - 2 ], treeArray[ j ] );
43     return treeArray[ 0 ];
44 }

```

图12-51 配对堆：两趟合并法

## 小结

本章中，我们看到二叉查找树几种有效的变体。自顶向下伸展树提供了 $O(\log N)$ 摊还性能，treap树给出 $O(\log N)$ 随机化的性能，而红黑树、确定性跳跃表和AA树均给出对基本操作的 $O(\log N)$ 最坏情形性能。在各种结构之间的权衡涉及代码复杂性、删除的简易性以及不同的查找和插入的开销。很难说哪种结构是明显的赢家。重复出现的论题包括树的旋转以及使用标记结点以避免对NULL指针的许多麻烦的测试，若不使用标记结点这些测试原本是必不可少的。尽管理论的界不是最优的， $k$ -d树还是提供了执行范围查找的实践方法。

最后，我们描述了配对堆并对配对堆进行了编程，它似乎是最实际的可合并的优先队列，特别是需要decreaseKey操作的时候。不过，在理论上它的效率不如斐波那契堆。

## 练习

- 12.1 证明：自顶向下伸展的摊还开销为 $O(\log N)$ 。
- \*\*12.2** 证明：对于自底向上伸展存在每次访问需要 $2 \log N$ 次旋转的访问序列。证明类似的结果对于自顶向下伸展也成立。
- 12.3 修改伸展树以支持对第 $k$ 个最小项的查询，这在确定性跳跃表中如何处理？
- 12.4 根据经验比较简化的自顶向下伸展和原始描述的自顶向下伸展。
- 12.5 编写红黑树的删除过程。
- 12.6 证明：红黑树的高度最多为 $2 \log N$ ，并且这个界实质上不能再降低。
- 12.7 证明：每一棵AVL树都可以被涂成红黑树。所有的红黑树都是AVL树吗？
- 12.8 证明：1-2-3确定性跳跃表可以表示成一棵2-3-4树（即4阶B树），它的项在内部结点和树叶上。
- 12.9 如果试图插入已经在确定性跳跃表中存在的项，那么会发生什么情况？
- 12.10 证明：在1-2-3确定性跳跃表中最多用到 $2N$ 个结点。
- \*12.11** 在C++语言中，可以把每一个抽象结点表示成动态分配的前向指针的数组而不是指针的链表，指出如何用这种方法实现1-2-3确定性跳跃表并保持每个操作的 $O(\log N)$ 时间界。
- 12.12 编写1-2-3确定性跳跃表的删除过程。
- 12.13 证明AA树中关于删除的算法是正确的。
- 12.14 给出AA树的一种非递归的自顶向下实现方法。将其与本书中的实现方法在简单性和效率方面进行比较。
- 12.15 递归地编写出skew过程和split过程，使得对删除操作每个过程只需调用一次。
- 12.16 AA树使用的程序代码比BB树少多少行？这能使AA树更快吗？
- 12.17 通过使用一个栈来非递归地实现treap树的插入例程。这种努力值得吗？
- 12.18 通过使用访问次数作为优先级并在每次访问后需要时执行旋转，可以使treap树成为自调整的。将这种方法和随机化方法进行比较。或者，在每次访问项 $X$ 时生成一个随机数。如果这个数小于 $X$ 当前的优先级，那么就用它作为 $X$ 的新优先级（执行相应的旋转）。
- \*\*12.19** 证明：如果把项排序，那么即使优先级并未排序，treap树也可以以线性时间构造。
- 12.20 不用nullNode标记实现某些树结构。使用这个标记可以节省多少编程工作？
- 12.21 假设对于每个结点我们把NULL链的个数存储在其子树中；称之为结点的权（weight）。采用下列方法：如果左子树和右子树的权相差超出因子2，那么彻底重建根在该结点的子树。证明下列结论：
- 可以以 $O(S)$ 重建一个结点，其中 $S$ 是该结点的权。
  - 该算法每次插入操作的摊还时间为 $O(\log N)$ 。
  - 可以以 $O(S \log S)$ 时间在 $k$ -d树中重建一个结点，其中 $S$ 是该结点的权。
  - 可以将该算法用于 $k$ -d树，其每次插入的代价为 $O(\log^2 N)$ 。
- 12.22 假设我们对任意一棵2-d树调用rotateWithLeftChild，详细解释其结果不再是一棵可用的2-d树的全部原因。
- 12.23 实现对于 $k$ -d树的插入和范围查询。不要使用递归。
- 12.24 对与 $k=3, 4, 5$ 对应的 $p$ 的值，确定部分匹配查询的时间。
- 12.25 对于一棵理想平衡 $k$ -d树，求出书中引用的一次范围查询（见12.6节）的最坏情形运行时间。
- 12.26 2-d堆是允许每一项拥有两个单个键的一种数据结构。deleteMin可以对于这两个键中的任一个执行。2-d堆是具有下述性质的完全二叉树：对于偶数深度上的任一结点 $X$ ，存储在 $X$ 上的项具有

它的子树上最小的#1键，而对于奇数深度上的任一结点 $X$ ，存储在 $X$ 上的项具有它的子树上最小的#2键。

- 画出关于(1, 10)、(2, 9)、(3, 8)、(4, 7)、(5, 6)诸项的一个可能的2-d堆。
- 如何找出具有最小#1键的项？
- 如何找出具有最小#2键的项？
- 给出一个将一新项插入到2-d堆中的算法。
- 给出一个对于任一键执行deleteMin操作的算法。
- 给出一个以线性时间完成buildHeap的算法。

12.27 将前面的练习推广以得出一个 $k$ -d堆，在这个堆中每一项都可有 $k$ 个单个键。你应该能够得到下面的界：以 $O(\log N)$ 完成insert，以 $O(2^k \log N)$ 完成deleteMin，以 $O(kN)$ 完成buildHeap。

12.28 证明： $k$ -d堆可以用于实现双端优先队列。

12.29 抽象地推广 $k$ -d堆，使得只有那些根据#1键分支的层有两个儿子（所有其他层都有一个儿子）。

- 需要指针吗？
- 显然，那些基本算法仍然有效；新的时间界是多少？

12.30 使用 $k$ -d树实现deleteMin。对于随机树，你期望其平均运行时间是多少？

12.31 使用 $k$ -d堆实现双端队列，该队列也支持deleteMin操作。

12.32 使用nullNode标记实现配对堆。

\*\*12.33 证明：对于本书中的配对堆算法，每次操作的摊还时间为 $O(\log N)$ 。

12.34 combineSiblings的另一种方法是把所有的兄弟都放到一个队列中，并反复dequeue及合并队列中的前两项，把结果放到队尾。实现这种方法。

12.35 在前面的练习中不用队列而使用栈是个不好的主意，通过给出一个序列使得每次操作花费 $\Omega(N)$ 来加以论证。这就是从左到右单趟合并。

12.36 不用decreaseKey可以除去父链。使用斜堆效果会如何？

12.37 设下列每一问都可以表示成一棵具有儿子和父亲指针的树。解释如何实现decreaseKey操作。

561

- 二叉堆。
- 伸展树。

12.38 当用图形观察时，2-d树上的每个结点都把平面划分成一些区域。例如，图12-52显示了对图12-42中的2-d树的前5次插入。第一次插入 $p_1$ 把平面分成左右两部分。第二次插入 $p_2$ 又将左部分分成上下两部分，依次类推。

- 给定 $N$ 项，它们插入的顺序是否影响最后的划分？
- 如果两个不同的插入序列得到相同的树，那么对应的划分是否相同？
- 给出经过 $N$ 次插入之后所划分的区域的个数的公式。
- 指出图12-42中的2-d树的最后的划分。

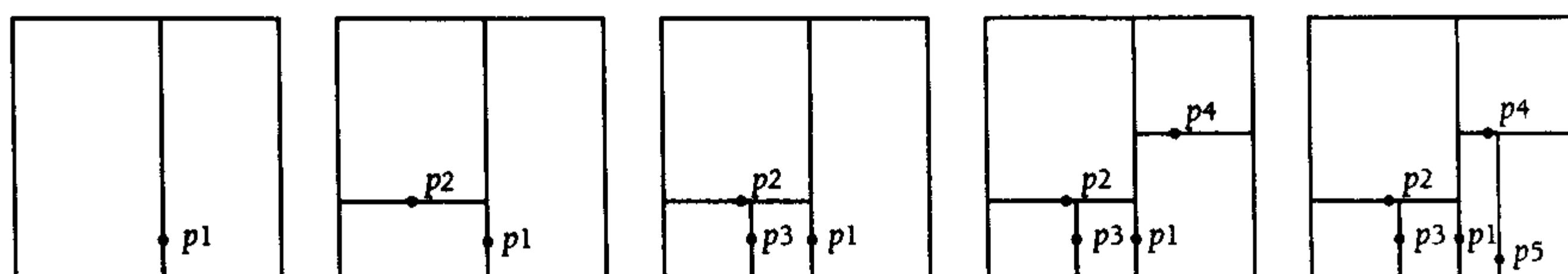


图12-52 由2-d树在插入 $p_1 = (53, 14)$ 、 $p_2 = (27, 28)$ 、 $p_3 = (30, 11)$ 、 $p_4 = (67, 51)$ 、 $p_5 = (70, 3)$ 后划分的平面

12.39 2-d树的一种变体是四叉树 (quad tree)。图12-53显示了平面是如何被一棵四叉树划分的。开始时有一个区域（它常常是一个方块，但不是必需的）。每个区域可存储一个点。如果将第2个点插入到区域中，那么区域就被划分成4块相等大小的象限（右上，右下，左下，左上）。如果能够把点



放在不同的象限中（如在 $p_2$ 插入时的情形），那么插入完成；否则，继续递归地分裂区域（就像插入 $p_5$ 时所做的那样）。

a. 给定 $N$ 项的集合，插入的顺序影响最后的划分吗？

b. 如果把图12-42的2-d树中的元素插入到四叉树中，指出最后的划分。

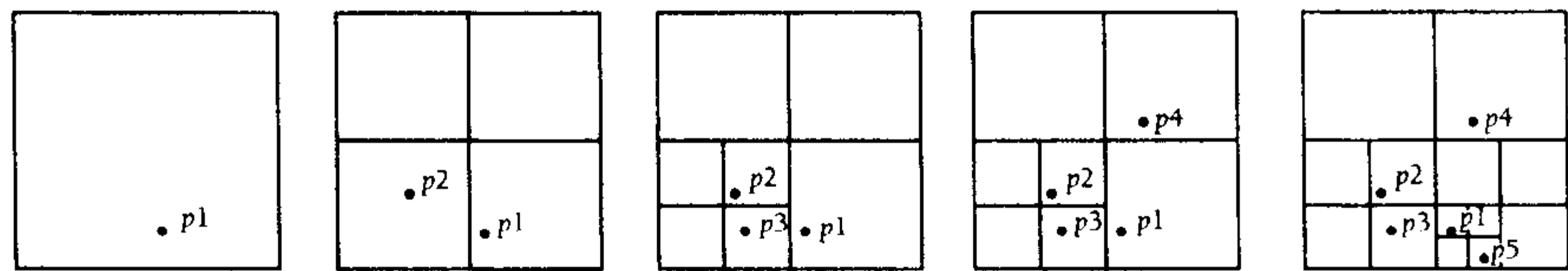


图12-53 由四叉树在插入 $p_1 = (53,14)$ 、 $p_2 = (27,28)$ 、 $p_3 = (30,11)$ 、 $p_4 = (67,51)$ 、 $p_5 = (70,3)$ 后划分的平面

562

12.40 树数据结构可以存储四叉树。我们保留原始区域的边界，树的根表示原始区域。每个结点或者是一片树叶（存放一个插入项），或者刚好有4个儿子（代表4个象限）。为了进行查找，从根开始并反复分支到相应的象限，直到到达一片树叶（或是null项）为止。

a. 画出对应图12-53的四叉树。

b. 哪些因素影响（四叉）树的深度？

c. 描述一种算法，使得在四叉树中执行一次正交范围查询。

参考文献

自顶向下伸展树在原始伸展树论文[29]中进行了描述。类似的但不使用重要的旋转的方法在[31]中描述。自顶向下红黑树算法取自[17]，更易于理解的描述可见于[28]。自顶向下红黑树不使用标记结点的实现在[14]中给出，它提供了nullNode实用性的令人信服的论证。确定性跳跃表及其变体在[23]和[26]中讨论。对称二叉B树来源于[6]；书中讨论的AA树的实现采用[1]和[3]中的描述。treap树[4]基于[32]中描述的笛卡儿树（Cartesian tree）。相关的数据结构是优先查找树（priority search tree）[21]。

$k$ -d树首先在[7]中介绍。其他的范围查找算法在[8]中描述。在平衡 $k$ -d树上范围查找的最坏情形在[19]中得到，而书中引用的平均情形结果来自[13]和[10]。

配对堆及练习中提出的一些变体在[16]中描述。论文[18]提出伸展树是在不需要decreaseKey操作时选择的优先队列。另外一篇论文[30]提出配对堆达到与斐波那契堆相同的渐近界但在实践中性能更好。然而，一篇使用优先队列实现最小生成树算法的相关论文[22]提出decreaseKey的摊还时间不是 $O(1)$ 。M.Fredman[15]通过证明存在使decreaseKey操作的摊还时间为次最优（事实上，最少为 $\Omega(\log \log N)$ ）的序列而解决了最优性问题。不过，他还证明了，当用来实现Prim最小生成树算法时，如果图稍微稠密（即图中边的条数为 $O(N^{1+\epsilon})$ ，其中 $\epsilon$ 为任意值），那么配对堆则是最优的。然而，配对堆的完整分析仍然尚未解决。

大部分练习的解可以在原始参考文献中找到。练习12.21代表多少有些流行的一种“懒惰”平衡方法。[20]、[5]、[11]和[9]描述了一些特殊的策略；[2]指出在一种框架内如何实现这些方法。满足练习12.21中的性质的树是加权平衡的（weight-balanced），这些树也可通过旋转保持其特性[24]，(d)问取自[25]。练习12.26到练习12.28的解可以在[12]中找到。对四叉树的描述见于[27]。

1. A. Andersson, "A Note on Searching a Binary Search Tree," *Software—Practice and Experience*, 21 (1991), 1125-1128.
2. A. Andersson, "General Balanced Trees," *Journal of Algorithms*, 30 (1999), 1-28.
3. A. Andersson, "Balanced Search Trees Made Simple," *Proceedings on the Third Workshop on*

- Algorithms and Data Structures* (1993), 61-71.
4. C. Aragon and R. Seidel, "Randomized Search Trees," *Proceedings of the Thirtieth Annual Symposium on Foundations of Computer Science* (1989), 540-545.
  5. J. L. Baer and B. Schwab, "A Comparison of Tree-Balancing Algorithms," *Communications of the ACM*, 20 (1977), 322-330.
  6. R. Bayer, "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms," *Acta Informatica*, 1 (1972), 290-306.
  7. J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, 18 (1975), 509-517.
  8. J. L. Bentley and J. H. Friedman, "Data Structures for Range Searching," *Computing Surveys*, 11 (1979), 397-409.
  9. H. Chang and S. S. Iyengar, "Efficient Algorithms to Globally Balance a Binary Search Tree," *Communications of the ACM*, 27 (1984), 695-702.
  10. P. Chanzy, "Range Search and Nearest Neighbor Search," *Master's Thesis*, McGill University (1993).
  11. A. C. Day, "Balancing a Binary Tree," *Computer Journal*, 19 (1976), 360-361.
  12. Y. Ding and M. A. Weiss, "The k-d Heap: An Efficient Multi-Dimensional Priority Queue," *Proceedings of the Third Workshop on Algorithms and Data Structures* (1993), 302-313.
  13. P. Flajolet and C. Puech, "Partial Match Retrieval of Multidimensional Data," *Journal of the ACM*, 33 (1986), 371-407.
  14. B. Flamig, *Practical Data Structures in C++*, John Wiley, New York (1994).
  15. M. Fredman, "Information Theoretic Implications for Pairing Heaps," *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing* (1998), 319-326.
  16. M. L. Fredman, R. Sedgwick, D. D. Sleator, and R. E. Tarjan, "The Pairing Heap: A New Form of Self-Adjusting Heap," *Algorithmica*, 1 (1986), 111-129.
  17. L.J. Guibas and R. Sedgwick, "A Dichromatic Framework for Balanced Trees," *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science* (1978), 8-21.
  18. D. W. Jones, "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *Communications of the ACM*, 29 (1986), 300-311.
  19. D. T. Lee and C. K. Wong, "Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees," *Acta Informatica*, 9 (1977), 23-29.
  20. W. A. Martin and D. N. Ness, "Optimizing Binary Trees Grown with a Sorting Algorithm," *Communications of the ACM*, 15 (1972), 88-93.
  21. E. McCreight, "Priority Search Trees," *SIAM Journal of Computing*, 14 (1985), 257-276.
  22. B. M. E. Moret and H. D. Shapiro, "An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree," *Proceedings of the Second Workshop on Algorithms and Data Structures* (1991), 400-411.
  23. J. I. Munro, T. Papadakis, and R. Sedgwick, "Deterministic Skip Lists," *Proceedings of the Third Annual Symposium of Discrete Algorithms* (1992), 367-375.
  24. J. Nievergelt and E. M. Reingold, "Binary Search Trees of Bounded Balance," *SIAM Journal on Computing*, 2 (1973), 33-43.
  25. M. H. Overmars and J. van Leeuwen, "Dynamic Multidimensional Data Structures Based on Quad and K-D Trees," *Acta Informatica*, 17 (1982), 267-285.
  26. T. Papadakis, *Skip Lists and Probabilistic Analysis of Algorithms*, Ph.D. Dissertation, University of Waterloo (1993).
  27. H. Samet, "The Quadtree and Related Hierarchical Data Structures," *Computing Surveys*, 16 (1984), 187-260.
  28. R. Sedgwick, *Algorithms in C++*, Parts 1-4 (3rd edition), Addison-Wesley, Reading, Mass. (1998).

29. D. D. Sleator and R. E. Tarjan, "Self Adjusting Binary Search Trees," *Journal of the ACM*, 32 (1985), 652-686.
30. J. T. Stasko and J. S. Vitter, "Pairing Heaps: Experiments and Analysis," *Communications of the ACM*, 30 (1987), 234-249.
31. C.J. Stephenson, "A Method for Constructing Binary Search Trees by Making Insertions at the Root," *International Journal of Computer and Information Science*, 9 (1980), 15-29.
32. J. Vuillemin, "A Unifying Look at Data Structures," *Communications of the ACM*, 23 (1980), 229-239.



## 类模板的分离编译

如前所示，若完全在声明中实现类模板，则仅需要很少的语法。由于许多过去支持类模板分离编译的编译器已经弱化并且平台特定化了，所以事实上现在很多类模板都是通过这种方式实现的。因此，在很多情况下，整个类连同其实现都放在一个头文件里。流行的标准库实现也是遵循这个策略来实现类模板。

现行的标准正试图矫正这种局面，因此，人们更多地关注如何将类模板的声明从其实现中分离出来。而这也成了编程语言的一部分。对模板和非模板该思想都是一样的。

图A-1给出了MemoryCell类模板的声明。当然，这部分是很简单的，因为它仅是整个类模板的一部分。

```
1 #ifndef MEMORY_CELL_H
2 #define MEMORY_CELL_H
3
4 /**
5  * A class for simulating a memory cell.
6  */
7 template <typename Object>
8 class MemoryCell
9 {
10     public:
11         explicit MemoryCell( const Object & initialValue = Object( ) );
12         const Object & read( ) const;
13         void write( const Object & x );
14
15     private:
16         Object storedValue;
17 };
18
19 #endif
```

图A-1 MemoryCell类模板接口

在实现中有许多函数模板。这意味着每个函数都必须包括模板声明，并且在使用作用域操作符的时候，类的名称必须通过模板变量来实例化。因此在图A-2中，类的名字是MemoryCell<Object>。虽然语法看上去无伤大雅，却使程序很繁琐。例如，不使用额外的代码就可以在说明中定义operator=。在实现中采用下面的蛮力代码<sup>1</sup>：

```
Template <typename Object>
Const MemoryCell <Object> &
MemoryCell<Object>::operator={ const MemoryCell<Object> & rhs}
```

1. 注意一些(在::后的)MemoryCell<Object>可以忽略,编译器将MemoryCell解释成MemoryCell<Object>。



```

{
    if( this!=&rhs )
        storedValue = rhs.storedValue;
    return *this;
}

```

567 至此，问题就变成了如何组织类模板声明和成员函数模板定义。主要的问题在于图A-2中的实现不是实际的函数，而是等待扩展的简单模板。但是，即使在MemoryCell模板被实例化的时候它们也没有扩展。每个成员函数模板只有在被调用的时候才扩展。

```

1 #include "MemoryCell.h"
2
3 /**
4  * Construct the MemoryCell with initialValue.
5  */
6 template <typename Object>
7 MemoryCell<Object>::MemoryCell( const Object & initialValue )
8     : storedValue( initialValue )
9 {
10 }
11
12 /**
13  * Return the stored value.
14  */
15 template <typename Object>
16 const Object & MemoryCell<Object>::read( ) const
17 {
18     return storedValue;
19 }
20
21 /**
22  * Store x.
23  */
24 template <typename Object>
25 void MemoryCell<Object>::write( const Object & x )
26 {
27     storedValue = x;
28 }

```

图A-2 MemoryCell类模板的实现

## A.1 头文件的内容

优先选择是将声明与实现都放到头文件中。这对类是行不通的，因为如果几个不同的源文件都有处理这个头文件的包含指令的话，就会出现重复定义函数的情况。但是，在这里的由于只是模板而不是真实的类，则不会有问题。

为阅读方便，可以在使用这个策略时让头文件发出一个包含指令（在#endif前）来自动读入实现文件。使用该策略时，存储模板的.cpp文件不能直接编译。

## A.2 显式实例化

568 对某些编译器，如果使用显示实例化则可以获得许多分离编译的优势。在这种情况下，将.h和.cpp文件作为一般的类来处理。这样一来，图A-1和图A-2的代码就与图中所示一致。头文件

没有读入实现的包含指令。主程序中只有一个包含该头文件的指令。图A-3是一个典型的测试程序。如果同时编译两个.cpp文件，就会发现没有找到实例化的成员函数。这个问题可以通过生成一个对我们所用的类型包含显示实例化MemoryCell的分离文件来解决。显示实例化的例子在图A-4中给出。该文件作为项目的一部分进行编译，而模板实现文件（图A-2）不作为项目的一部分编译。我们已经对数个较老的编译器成功使用了该技术。不利的方面是所有的模板扩展都必须由编程者列出来。而且，如果类模板使用了其他类模板的话，那些被调用的模板有时候也需要列出来。有利的方面在于如果MemoryCell中的成员函数的实现发生变化，只有MemoryCell-Expand.cpp需要重新编译。

569

```

1 #include "MemoryCell.h"
2 #include <iostream>
3 using namespace std;
4
5 int main( )
6 {
7     MemoryCell<int>    m1;
8     MemoryCell<double> m2( 3.14 );
9
10    m1.setValue( 37 );
11    m2.setValue( m2.getValue( ) * 2 );
12
13    cout << m1.getValue( ) << endl;
14    cout << m2.getValue( ) << endl;
15
16    return 0;
17 }
```

图A-3 MemoryCell类模板的使用

```

1 #include "MemoryCell.cpp"
2
3 template class MemoryCell<int>;
4 template class MemoryCell<double>;
```

图A-4 实例化文件MemoryCellExpand.cpp

## A.3 导出指令

新编译器支持导出指令（export directive）。这是现行语言新加的。如图A-5所示，如果在类模板声明之前发出导出指令，那么，就不再需要在A.2节所添加的额外文件，并且程序可以正确链接。但是不能期望在所有的编译器上它都能正确实现。导出指令对函数模板也适用。

570

```

1 #ifndef MEMORY_CELL_H
2 #define MEMORY_CELL_H
3
4 template <typename Object>
5 export class MemoryCell
6 {
7     :
8 }
```

图A-5 导出指令

# 索引

索引中的页码为英文原书的页码，与书中边栏的页码一样。

## A

- AA-trees (AA树), 540~545
- AANode structure (AANode结构), 541
- AATree class (AATree类), 541
- Abramson, B., 486
- Abstract data types (ADTs) (抽象数据类型), 71
  - disjoint sets (不相交集), 见Disjoint Sets
  - hash tables (散列表), 见Hash tables
  - lists (表), 见Lists
  - queues (队列), 见Queues
  - stacks (栈), 见Stacks
- Abstract skip list representation (抽象跳跃表表示), 537
- Accessors (访问函数)
  - for classes (类), 15
  - for matrices (矩阵), 37
- Ackermann function (Ackermann函数), 325~326
- Activation records (活动记录), 102
- Activity-node graphs (动作-节点图), 361~362
- Acyclic graphs (无环图), 339
  - in shortest-path algorithms (最短路径算法), 360~364, 483~484
  - topological sorts for (拓扑排序), 342~345
- Address-of operator (&) for pointers (指针的地址运算符 (&)), 21, 295
- Adelson-Velskii, G.M., 182
- Adjacency lists (邻接表)
  - for graph representation (图表示法), 341~342
  - references for (引用), 404
- Adjacency matrices (邻接矩阵), 340~341
- Adjacent vertices (邻接顶点), 339
- Aggarwal, A., 486
- Agrawal, M., 486
- Aho, A.V., 69, 182, 336, 405
- Ahuja, R.K., 405
- Airport systems, graphs for (机场系统的图), 340
- Albertson, M.O., 41
- Algorithm analysis (算法分析), 43
  - components of (组成部分), 46~49
  - computational models for (计算模型), 46
  - mathematical background for (数学背景知识), 43~46
  - running times (运行时间), 见Running times
- Algorithm design (算法设计)
  - approximate bin packing (近似装箱), 见Approximate bin packing
  - packing
  - backtracking algorithms (回溯算法), 见Backtracking algorithms
  - divide and conquer strategy (分治策略), 见Divide and conquer strategy
  - dynamic programming (动态规划), 见Dynamic programming
  - greedy algorithms (贪心算法), 见Greedy algorithms
  - randomized algorithms (随机化算法), 454~455
- primality tests (素性测试), 461~464
- random number generators (随机数生成器), 455~459
- skip lists (跳跃表), 459~461
- All-pairs shortest-path algorithm (所有点对最短路径算法), 364, 451~454
- Allen, B., 182
- allPairs function (allPairs函数), 453~454
- Alon, N., 486
- Alpha-beta pruning ( $\alpha$ - $\beta$ 裁剪), 472~476, 486
- Amortized analysis (摊还分析), 491
  - binomial queues (二项队列), 492~497
  - Fibonacci heaps (斐波那契堆), 见Fibonacci heaps
  - references for (引用), 515
  - skew heaps (斜堆), 497~499
  - splay trees (伸展树), 149, 509~513
- Amortized bound times (摊还时间界), 491
- Ancestors in trees (树中的祖先), 114
- Andersson, A., 564
- Approximate bin packing (近似装箱), 419~420
  - best fit for (最佳适配), 423~424
  - first fit for (首次适配), 422~423
  - next fit for (下项适配), 421~422
  - off-line algorithms for (脱机算法), 424~427
  - on-line algorithms for (联机算法), 420~421
  - references for (引用), 485
- Approximate pattern matching (近似字型匹配), 482~483
- Aragon, C., 564
- Arcs, graph (图的弧), 339
- Arithmetic problems (运算问题)
  - integer multiplication (整数乘法), 438~439
  - matrix multiplication (矩阵乘法), 439~442
- Arithmetic series (算术级数), 5
- Arrays (数组)
  - for binary heaps (二叉堆), 216
  - C-style (C风格), 26~29

- for complete binary trees (完全二叉树), 215
- for graph representation (图表示), 340
- for hash tables (散列表), 见Hash tables
- inversion in (逆序), 265~266
- for lists (表), 72
- maps for (图), 168~173
- for matrices (矩阵), 见Matrices
- for queues (队列), 104~106
- for quicksort (快速排序), 284~285
- for skip lists (跳跃表), 540
- in sorting (排序), 见Sorting
- for stacks (栈), 96
- strings for (字符串), 17~18
- for vectors (向量), 80
- Articulation points (割点), 381~385
- ASCII character set (ASCII字符集), 414
- assertIsValid function (assertIsValid函数), 94
- Assigning pointers (指针赋值), 20
- assignLow function (assignLow函数), 383~384
- Assignment operators (赋值运算符), 24~26
- assignNum function (assignNum函数), 383~384
- at function (at函数), 75
- Atkinson, M.D., 258
- auf der Heide, F. Meyer, 212
- Augmenting paths (增长通路), 367~372
- Automatic variables (自动变量), 20
- Average-case analysis (平均情形分析), 47
  - binary search trees (二叉查找树), 133~136
  - quicksorts (快速排序), 288~290
- AVL trees (AVL树), 136~139
  - double rotation (双旋转), 141~149
  - references for (引用), 181~182
  - single rotation (单旋转), 139~141
- AvlNode structure (AvlNode结构), 146~148
- B**
- B-trees (B树)
  - with disk access (磁盘访问), 159~164
  - references for (引用), 182
- Back edges in depth-first searches (深度优先搜索中的后向边), 389
- back function (back函数)
  - for lists (表), 75, 87
  - for stacks (栈), 96
  - for vectors (向量), 75, 80, 82~83
- Backtracking algorithms (回溯算法)
  - games (博弈), 469
- alpha-beta pruning ( $\alpha$ - $\beta$ 裁剪), 472~476
- minimax algorithm (极小极大算法), 469~472
  - principles (原理), 464
  - turnpike reconstruction (收费公路重建), 465~469
- Bacon numbers (Bacon数), 403
- Baer, J.L., 564
- Baeza-Yates, R.A., 182~183, 212, 312
- Balance conditions (平衡条件), 136
  - AVL trees (AVL树), 136~139
- double rotation (双旋转), 141~149
- references for (引用), 181~182
- single rotation (单旋转), 139~141
- Balanced binary search trees (平衡二叉查找树), 167~168
- Balancing symbols, stacks for (栈的平衡符号), 96~97
- Banachowski, L., 336
- Base 2 logarithms (以2为底的对数), 3
- Base cases (基准情形)
  - induction (归纳法), 6
  - recursion (递归法), 8~9, 11
- Baseball card collector problem (棒球卡收藏家问题), 404
- Bavel, Z., 41
- Bayer, R., 182, 564
- BB-trees (binary B-trees) (二叉B树), 540
- begin function (begin函数)
  - for iterators (迭代器), 76, 79~80, 82
  - for lists (表), 86, 93
  - for sets (集合), 165
- Bell, T., 212, 486
- Bellman, R.E., 405, 486
- Bentley, J.L., 69, 182, 312, 486, 564
- Best-case analysis (最佳情形分析), 47, 288
- Best fit bin packing algorithm (最佳适配装箱算法), 423~424
- bestMove function (bestMove函数), 471
- Bhaskar, S., 487
- Biconnected graphs (双连通图), 381~385, 404
- Big-Oh notation (大O记法), 43~46, 159~160
- Big-Omega notation (大 $\Omega$ 记法), 43~46
- Bin packing (装箱), 396, 419~420
  - best fit for (最佳适配), 423~424
  - first fit for (首次适配), 422~423
  - next fit for (下项适配), 421~422
  - off-line algorithms for (脱机算法), 424~427
  - on-line algorithms for (联机算法), 420~421
  - references for (引用), 485
- Binary B-trees (BB-trees) (二叉B树), 540
- Binary heaps (二叉堆), 215
  - heap order (堆序), 216~217
  - operations (操作)
- buildHeap, 223~225
- decreaseKey, 222
- deletekey, 222
- deleteMin, 218~221
- increaseKey, 222
- insert, 217~219



- remove, 222
    - references for (引用), 257
    - structure (结构), 215~216
  - Binary search trees (二叉查找树), 113, 119~120, 124~125
    - average-case analysis (平均情形分析), 133~136
    - AVL trees (AVL树), 136~139
  - double rotation (双旋转), 141~149
  - references for (引用), 181~182
  - single rotation (单旋转), 139~141
    - class template for (类模板), 124~129
    - operations (操作)
  - contains (容器), 125
  - destructors and copy assignment (析构函数和复制赋值), 132~133
  - findMin and findMax (findMin和findMax), 125
  - insert, 129~130
  - remove, 130~132
    - optimal (最优), 447~451, 485
    - for priority queue implementation (优先队列实现), 214
    - red-black (红-黑), 见Red-black trees
    - references for (引用), 181
  - Binary searches (二分搜索), 58~60
  - Binary trees (二叉树)
    - expression (表达式), 121~124
    - Huffman code (赫夫曼编码), 414~419
    - implementations (实现), 120~121
    - traversing (遍历), 158~159
  - BinaryHeap class (BinaryHeap类), 216
  - BinaryNode structure (BinaryNode结构)
    - binary search trees (二叉查找树), 126
    - binary trees (二叉树), 120
    - top-down splay trees (自顶向下伸展树), 521~522
  - binarySearch function (binarySearch函数), 59~60
  - BinarySearchTree class (BinarySearchTree类), 124, 126~128
  - Binomial queues (二项队列), 239
    - amortized analysis (摊还分析), 492~497
    - implementation (实现), 244~250
    - lazy merging for (懒惰合并), 502~506
    - operations (操作), 241~244
    - references for (引用), 257, 515
    - structure of (结构), 240
  - Binomial trees (二项树), 240, 492
  - BinomialNode structure (BinomialNode结构), 246
  - BinomialQueue class (BinomialQueue类), 246~247
  - Bipartite graphs (二分图), 398
  - Bitner, J.R., 182
  - Bloom, G.S., 486
  - Blum, M., 486
  - Blum, N., 337
  - Boggle game (Boggle游戏), 484
  - Bollobas, B., 336
  - Bookkeeping costs in recursion (递归中的簿记开销), 11
  - Borodin, A., 486
  - Boruvka, O., 405
  - Bottom-up insertion in red-black trees (红黑树中的自底向上插入), 526~527
  - Bound times, amortized (摊还时间界), 491
  - Bounds of function growth (函数增长界), 44
  - Boyer, R.S., 212
  - Braces({}), balancing (花括号平衡), 96~97
  - Brackets([]) (方括号)
    - balancing (平衡), 96~97
    - operator (运算符), 见Operators
  - Breadth-first searches (广度优先搜索), 349~351
  - Bright, J.D., 258
  - Brodal, G.S., 258
  - Brown, D.J., 488
  - Brown, M.R., 258, 515
  - Brualdi, R.A., 41
  - Bucket sorts (桶排序), 299~300
  - buildBinomialQueue function (buildBinomialQueue函数), 495
  - buildHeap function (buildHeap函数)
    - for binary heaps (二叉堆), 216, 223~226
    - for heapsorts (堆排序), 271
    - for Huffman algorithm (赫夫曼算法), 419
    - for leftist heaps (左式堆), 233
- ## C
- C-style arrays and strings (C风格数组和字符串), 26~29
  - Calls, stacks for (调用栈), 102~104
  - capacity function (capacity函数)
    - for binomial queues (二项队列), 246
    - for vectors (向量), 75, 82~83
  - Carlsson, S., 258
  - Carmichael numbers (Carmichael数), 462
  - Carter, J.L., 212
  - Cartesian trees (笛卡儿树), 563
  - Cascading cuts for Fibonacci heaps (斐波那契堆的级联切除), 506~509
  - CaseInsensitiveCompare class (CaseInsensitiveCompare类), 35~36, 166
  - Catalan numbers (Catalan数), 446
  - Chaining for hash tables (散列表的链接), 188~192
  - Chang, H., 564
  - Chang, L., 486
  - Chang, S.C., 258
  - change function (change函数), 78
  - Chanzy, P., 564
  - Character sets (字符集), 414
  - Character substitution problem (字符替换问题), 168~173

- Characters, arrays of (字符数组), 27
- Checkers (检查程序), 471
- Chen, J., 258
- Cheriton, D., 258, 405
- Cheriyān, J., 405
- Chess (棋), 471, 484
- Children in trees (树中子结点(儿子)), 113~115
- Christofides, N., 487
- Circle packing problem (装圆问题), 480
- Circular arrays (循环数组), 105
- Circular linked lists (循环链表), 111
- Circular logic, recursion as (循环逻辑递归), 8
- Classes and class templates (类和类模板), 11, 30~32
  - accessors for (访问函数), 15
  - compilation (编译), 35~37, 567~568
- explicit instantiation (显式实例化), 568~570
- export directive (导出指示), 570
- for header information (头文件信息), 568
  - constructors for (构造函数), 12~15, 24
  - defaults for (默认值), 24~27
  - destructors for (析构函数), 23~24
  - interface/implementation separation in (接口/实现分离), 15~17
  - string and vector (string和vector), 17~19
  - syntax for (语法), 12
- clear function (clear函数)
  - for containers (容器), 75
  - for lists (列表), 85~86
  - for priority queues (优先队列), 251
  - for vectors (向量), 80
- Cleary, J.G., 486
- Clique problem (团问题), 396, 404
- Clocks (时钟)
  - for event simulation (事件模拟), 227
  - for random numbers (随机数), 456
- clone function (clone函数)
  - for binary search trees (二叉查找树), 134
  - for binomial queues (二项队列), 246
  - for leftist heaps (左式堆), 234
  - for red-black trees (红黑树), 529
  - for top-down splay trees (自顶向下伸展树), 521
- Closest points problem (最近点问题)
  - divide and conquer strategy (分治策略), 430~435
  - references for (引用), 485
- Clustering in hash tables (散列表聚集), 193~195, 199
- Code bloat (代码膨胀), 29
- Coin changing problem (钱币兑换问题), 352, 410, 483
- Collections (集合), 74~75
- Collisions in hashing (散列冲突), 186~188, 192
  - double hashing (双散列), 199~200
  - linear probing (线性探测), 193~195
  - quadratic probing (平方探测), 195~199
- Coloring graphs (着色图), 396
- combineSiblings function (combineSiblings函数), 555, 558~559
- combineTrees function (combineTrees函数), 246~247
- Comer, D., 182
- Commands, preprocessor (预处理指令), 15~16
- Comparable objects (可比对象), 32~33
  - for binary search trees (二叉查找树), 124
  - sorting (排序), 292~296
- compareAndLink function (compareAndLink函数), 555~556
- Comparison-based sorting (基于比较的排序), 261
- Comparisons (比较)
  - arrays (数组), 17
  - with iterators (迭代器), 77
  - pointers (指针), 20
  - in selection problem (选择问题), 437~438
  - strings (字符串), 18
- Compilation (编译), 35~37, 567~568
  - explicit instantiation in (显式实例化), 568~570
  - export directive in (导出指示), 570
  - hash tables for (散列表), 207
  - for header information (头文件信息), 568
- Complete binary trees (完全二叉树), 215~217
- Complete graphs (完全图), 340
- Compound interest rule in recursion (递归中的合成效益法则), 11
- Compression (压缩)
  - file (文件), 413~419
  - path (路径), 324~331
- Computational geometry (计算几何)
  - in divide and conquer strategy (分治策略), 428
  - turnpike reconstruction (收费公路重建), 465~469
- Computational models (计算模型), 46
- computeAdjacentWords function (computeAdjacentWords函数), 170~172
- Congruential generators (同余数生成器), 456~459
- Connected graphs (连通图), 339~340
- Connectivity (连通性)
  - biconnected graphs (双连通图), 381~385
  - electrical (电气的), 315
  - undirected graphs (无向图), 339~340
- Consecutive statements in running time (连续语句的运行时间), 50
- const keyword (const关键字), 15
- const\_iterators, 77~79, 84~85, 88~89
- Constant growth rate (常数增长率), 45
- Constant member functions (常量成员函数), 15
- Constant reference (常量引用)

- parameter passing by (传递参数), 21
  - returning values by (返回值), 22
  - Constructors (构造函数), 12
    - copy (复制), 24
    - default parameters (默认参数), 12~13
    - explicit (显式), 14~15
      - initializer Lists (初始化列表), 13~14
    - for iterators (迭代器), 77
    - for matrices (矩阵), 37, 39
  - Containers (容器), 74~75, 77, 见Lists, Queues, Stacks
  - contains function (contains 函数)
    - for binary search trees (二叉查找树), 125~128
    - for binary searches (二分搜索), 59
    - for hash tables (散列表), 189, 191, 197~198
    - for skip lists (跳跃表), 539
    - for top-down splay trees (自顶向下伸展树), 521
  - continue function (continue函数), 124
  - Contradiction, proofs by (反证法证明), 7
  - Convergent series (收敛级数), 4
  - Conversions (转换)
    - infix to postfix (中缀到后缀), 99~102
    - for pointers (指针), 295~296
    - type (类型), 14
  - Convex hulls (凸包), 481~482
  - Convex polygons (凸多边形), 481
  - Cook, S., 405
  - Cook's theorem (Cook定理), 396
  - Copies for vectors (向量复制), 80
  - Coppersmith, D., 487
  - Copy assignment operators (复制赋值运算符), 24~26
    - for binary search trees (二叉查找树), 132~133
    - for matrices (矩阵), 39
  - Copy constructors (复制构造函数), 24
    - for iterators (迭代器), 77
    - for matrices (矩阵), 39
  - copy function (copy函数), 38
  - Copying (复制)
    - in indirect sorting (间接排序), 292~294
    - matrices (矩阵), 38
    - shallow and deep (深复制与浅复制), 25~26
  - Costs, graph edges (图边值), 339
  - Counterexample, proofs by (反例证明), 7
  - Counters in mergesorts (归并排序中的计数器), 274~275
  - Crane, C.A., 258
  - Critical paths (关键路径), 361~362, 364
  - Cross edges (交叉边), 389~390
  - Cryptography (密码学)
    - gcd algorithm in (gcd算法), 64
    - prime numbers for (素数), 461~462
  - Cubic growth rate (三次增长率), 45
  - Culberson, J., 182
  - Culik, K., 182
  - Current nodes for lists (列表中的当前节点), 84
  - Cutting nodes in leftist heaps (切除左式堆中的结点), 500~502
- ## D
- d*-heaps (*d*堆), 228~229, 257
  - DAGs (directed acyclic graphs) (有向无环图), 339, 342~345
  - Data members for matrices (矩阵的数据成员), 37
  - Day, A.C., 564
  - Deap queues (双端优先队列), 257
  - Decision trees (决策树)
    - for lower bounds (下界), 297~299
    - references for (引用), 312
  - Declaring (声明)
    - objects (对象), 16~17
    - pointers (指针), 19
  - decreasekey function (decreasekey函数)
    - for binary heaps (二叉堆), 222
    - for binomial queues (二项队列), 247
    - for Dijkstra's algorithm (Dijkstra算法), 359
    - for Fibonacci heaps (斐波那契堆), 500~503, 506~509
    - for pairing heaps (配对堆), 553~555, 557
  - Deep copies (深复制)
    - vs. shallow (与浅复制), 25
    - for vectors (向量), 80
  - Defaults (默认)
    - for constructor parameters (构造函数参数), 12~13
    - problems with (问题), 24~27
  - Definitions, recursion in (递归定义), 9
  - delete function (delete函数)
    - for binary search trees (二叉查找树), 132~133
    - for lists (列表), 92
  - Delete operations (删除操作)
    - 2-d trees (2-d树), 551
    - AA-trees (AA树), 542~547
    - AVL trees (AVL树), 148
    - B-trees (B树), 164
    - binary heaps (二叉堆), 216, 218~222, 226
    - binary search trees (二叉查找树), 124, 126~127, 130~133
    - binomial queues (二项队列), 243~244, 246~247, 249~250, 492
    - d*-heaps (*d*堆), 228
    - destructors with (析构函数), 23~24
    - Fibonacci heaps (斐波那契堆), 500, 503~506, 508~509
    - hash tables (散列表), 189, 191, 196~197, 199
    - heapsorts (堆排序), 270
    - leftist heaps (左式堆), 233~234, 255
    - linked lists (链表), 73~74
    - lists (表), 72, 92, 109~110
    - multiway merges (多路合并), 302



- pairing heaps (配对堆), 553~555, 558
- pointers (指针), 20
- priority queues (优先队列), 213~214
- red-black trees (红黑树), 531, 534
- sets (集), 166
- skew heaps (斜堆), 237, 499
- skip lists (跳跃表), 535
- splay trees (伸展树), 156, 521, 524
- treaps (treaps树), 550
- vectors (向量), 80
- deleteKey function (deleteKey函数), 222
- deleteMax function (deleteMax函数), 271~272, 274
- deleteMin function (deleteMin函数)
  - binary heaps (二叉堆), 216, 218~221, 226
  - binomial queues (二项队列), 243~244, 246~247, 249~250
  - d-heaps (d堆), 228
  - Dijkstra's algorithm (Dijkstra算法), 358~360
  - Fibonacci heaps (斐波那契堆), 500, 503~506, 508~509
  - heapsorts (堆排序), 270
  - Huffman algorithm (赫夫曼算法), 419
  - Kruskal's algorithm (Kruskal算法), 378
  - leftist heaps (左式堆), 233~234
  - multiway merges (多路合并), 302
  - pairing heaps (配对堆), 553~555, 558
  - priority queues (优先队列), 213~214
  - skew heaps (斜堆), 237, 499
- Demers, A., 487
- Dense graphs (稠密图), 341, 364, 376
- Deo, N., 405
- Depth-first searches (深度优先搜索), 378~379
  - biconnected graphs (双连通图), 381~385
  - directed graphs (有向图), 388~390
  - Euler circuits (欧拉回路), 385~388
  - for strong components (强分支), 390~391
  - undirected graphs (无向图), 379~381
- Depth of trees (树的深度), 114
- Dequeues with heap order (具有堆序的双端队列), 514
- Dequeue operations (出队操作), 104~106, 110
- Dereferencing pointers (解引用指针), 295
- Descendants in trees (树中的后裔), 114
- Design rule in recursion (递归中的设计法则), 11
- Destructors (析构函数), 23~24
  - for binary search trees (二叉查找树), 132~133
  - for matrices (矩阵), 39
- Deterministic skip lists (确定性跳跃表), 535~539
- dfs function (dfs函数), 379
- Diamond dequeues (双端优先队列), 257
- Dictionaries, recursion in (词典递归), 9
- Dietzfelbinger, M., 212
- Digraphs (有向图), 339~340
  - all-pairs shortest paths in (所有点对最短路径), 451~454
  - depth-first searches (深度优先搜索), 388~390
  - representation of (表示), 340~342
- Dijkstra, E. W., 41, 405
- dijkstra function (dijkstra函数), 359
- Dijkstra's algorithm (Dijkstra算法), 351~360
  - for all-pairs shortest paths (所有点对最短路径), 451~452
  - and Prim's algorithm (Prim算法), 373~374
  - time bound improvements for (时间界改进), 499~500
- Dimensions for k-d trees (k-d树的维数), 553
- Diminishing increment sorts (缩减增量排序), 见Shellsort
- Ding, Y., 258, 564
- Dinic, E. A., 405
- Directed acyclic graphs (DAGs) (有向无环图), 339, 342~345
- Directed edges (有向边), 113
- Directed graphs (有向图), 339~340
  - all-pairs shortest paths in (所有点对最短路径), 451~454
  - depth-first searches (深度优先搜索), 388~390
  - representation of (表示), 340~342
- Directories (目录)
  - in extendible hashing (可扩散列), 204~205
  - trees for (树), 115~119
- Disjoint sets (不相交集), 315
  - dynamic equivalence problem in (动态等价问题), 316~317
  - equivalence relations in (等价关系), 315
  - for maze generation (迷宫生成), 331~334
  - path compression for (路径压缩), 324~325
  - references for (引用), 336~337
  - smart union algorithms for (灵巧求并运算), 321~323
  - structure of (结构), 317~321
  - worst case analysis for union-by-rank approach (按秩求并方法的最坏情形分析), 325~331
- DisjSets class (DisjSets类), 319~320, 323~325
- Disk I/O operations (磁盘I/O操作)
  - in extendible hashing (可扩散列), 204~205
  - and running times (运行时间), 159~160
  - in sorting (排序), 261
- Distances, closest points problem (最近点问题中的距离), 430~435
- Divide and conquer strategy (分治策略), 427
  - closest points problem (最近点问题), 430~435
  - components (组成部分), 427~428
  - integer multiplication (整数乘法), 438~439
  - matrix multiplication (矩阵乘法), 439~442
  - maximum subsequence sum (最大子序列和), 54~58
  - in mergesorts (归并排序), 275~279
  - quicksort (快速排序), 见Quicksort
  - running time of (运行时间), 428~430
  - selection problem (选择问题), 435~438
- Double hashing (双散列), 199~200, 211
- Double rotation operations (双旋转操作), 141~149



doubleWithLeftChild function (doubleWithLeftChild函数), 148~149  
 Doubly linked lists (双向链表), 74,84~85  
 Doyle, J., 337  
 drand48 function (drand48函数), 459  
 Dreyfus, S.E., 486  
 Driscoll, J. R., 258  
 DSL class (DSL类), 538~539  
 Du, H. C., 212  
 Dual-direction type conversions (双向类型转换), 296  
 Due, M. W., 258  
 dumpContents function (dumpContents函数), 252  
 Duplicate elements in quicksorts (快速排序中的元素复制), 283  
 Dynamic equivalence problem (动态等价问题), 316~317  
 Dynamic objects (动态对象), 19~20  
 Dynamic programming (动态规划), 442  
   all-pairs shortest path (所有点对最短路径), 451~454  
   optimal binary search trees (最优二叉查找树), 447~451, 485  
   ordering matrix multiplications (矩阵乘法次序), 444~447  
   references for (引用), 485  
   tables vs. recursion (表与递归), 442~444

## E

Eckel, B., 41  
 Edelsbrunner, H., 487  
 Edges (边)  
   in depth-first searches (深度优先搜索), 389~390  
   graph (图), 339~341  
   tree (树), 113  
 Edmonds, J., 405  
 Eight queens problem (八皇后问题), 483  
 Eisenbath, B., 182  
 Electrical connectivity (电气连通性), 315  
 Empirical running time confirmation (经验运行时间确定), 62~63  
 Employee class (Employee类), 32~33  
 empty function (empty函数)  
   for containers (容器), 75  
   for maps (图), 166  
   for priority queues (优先队列), 251  
   for sets (集), 165  
   for vectors (向量), 80,82~83  
 Empty lists (空表), 72  
 Enbody, R.J., 212  
 end function (end函数)  
   for iterators (迭代器), 76,79~80,82  
   for lists (表), 86  
   for maps (图), 166  
   for sets (集), 165

#endif preprocessor (#endif预处理指令), 16  
 Enqueue operations (入队操作), 104~106  
 Eppinger, J.L., 182  
 Eppstein, D., 487  
 Equivalence in disjoint sets (不相交集等价), 315~317  
 erase function (erase函数)  
   for iterators (迭代器), 77  
   for lists (表), 85,87,92~93  
   for maps (图), 166  
 Erase operations (去除操作), 见Delete operations  
 Eriksson, P., 259  
 Euclidean distance (欧几里得距离), 430  
 Euclid's algorithm running time (欧几里得算法的运行时间), 60~61  
 Euler circuits (欧拉回路), 385~388,392,400  
 Euler's constant (欧拉常数), 5, 289  
 eval function (eval函数), 444~445  
 Even, S., 405  
 Event-node graphs (事件节点图), 362~363  
 Event simulation (事件模拟), 227~228  
 Explicit constructors (显式构造函数), 14~15  
 Explicit instantiation (显式实例化), 568~570  
 Exponential growth rate (指数增长率), 45,52  
 Exponents and exponentiation (指数和幂运算)  
   formulas for (公式), 3  
   running time for (运行时间), 61~62  
 export directive (导出指令), 570  
 Expression trees (表达式树), 121~124  
 Extendible hashing (可扩散列), 204~205,211  
 External sorting (外部排序), 300  
   algorithm (算法), 301~302  
   model for (模型), 300  
   need for (需求), 300  
   references for (引用), 312  
   replacement selection in (替换选择), 304~305

## F

Factorials, recursion for (阶乘递归), 51  
 Fagin, R., 212  
 Fermat's lesser theorem (费马最小定理), 462  
 fib function (fib函数), 443  
 fibonacci function (fibonacci函数), 443  
 Fibonacci heaps (斐波那契堆), 499~500  
   cutting nodes in leftist heaps (切除左式堆中的结点), 500~502  
   for Dijkstra's algorithm (Dijkstra算法), 360  
   lazy merging for binomial queues (二项队列中的懒惰合并), 502~506  
   operations (操作), 506  
   for priority queues (优先队列), 258  
   time bound proof for (时间界证明), 506~509

- Fibonacci numbers (斐波那契数)
    - in polyphase merges (多相合并), 304
    - proofs for (证明), 6
    - recursion for (递归), 51~52, 442~445
  - File compression (文件压缩), 413~419
  - File servers (文件服务器), 107
  - Find operations (查找操作), 参见 Searches
    - biconnected graphs (双连通图), 385
    - binary heaps (二叉堆), 216~217
    - binary search trees (二叉查找树), 125~126, 128~129
    - binomial queues (二项队列), 246, 250
    - disjoint sets (不相交集), 316~317, 319~321, 324~325
    - hash tables (散列表), 197~199
    - lists (表), 72~73
    - maps (图), 166
    - shortest-path algorithms (最短路径算法), 365~366
    - top-down splay trees (自顶向下伸展树), 521
  - findArt function (findArt函数), 385
  - findChain function (findChain函数), 365~366
  - findCompMove function (findCompMove函数), 471, 473~475
  - findHumanMove function, (findHumanMove函数), 471~475
  - findKth operations (findKth操作), 72~73
  - findMax functions (findMax函数), 22
    - for binary search trees (二叉查找树), 125~126, 128~129
    - for function objects (函数对象), 34~36
    - template for (模板), 29~30
    - for top-down splay trees (自顶向下伸展树), 521
  - findMaxWrong function (findMaxWrong函数), 22
  - findMin function (findMin函数),
    - for binary heaps (二叉堆) 216~217
    - for binary search trees (二叉查找树), 125~126, 128~129
    - for binomial queues (二项队列), 246
    - for leftist heaps (左式堆), 234
    - for top-down splay trees (自顶向下伸展树), 521
  - findMinIndex function (findMinIndex函数), 246, 250
  - findNewVertexOfIndegreeZero function (findNewVertexOfIndegreeZero函数), 342~344
  - findPos function (findPos函数), 197~199
  - First fit algorithm (首次适配算法), 422~423
  - First fit decreasing algorithm (首次适配递减算法), 424~425
  - Fischer, M.J., 337, 489
  - Flajolet, P., 183, 212, 564
  - Flaming, B., 564
  - flip function (flip函数), 479
  - Floyd, R.W. 258, 312, 486~487
  - For loops in running time (for循环运行时间), 50
  - Ford, L.R., 312, 405
  - Forests (森林)
    - for binomial queues (二项队列), 240
    - in depth-first spanning (深度优先生成), 381
    - for disjoint sets (不相交集), 318
    - for Kruskal's algorithm (Kruskal算法), 376
  - Forward edges (前向边), 389
  - Four-parameter insertion sorts (四参数插入排序), 263
  - Fredman, M.L., 212, 258, 337, 405, 487, 515, 564
  - Friedman, J.H., 564
  - friend declarations (友元声明), 88
  - front function (front函数), 75, 87
  - Fulkerson, D.R., 405
  - Full nodes (满结点), 175
  - Full trees (满树), 415
  - Fuller, S.H., 183
  - Function objects (函数对象), 34~35
  - Function templates (函数模板), 29~30
  - Functions and function calls (函数和函数调用)
    - member (成员), 12
    - recursive (递归), 7~11
    - stacks for (栈), 102~104
- ## G
- Gabow, H.N., 258, 337, 405
  - Gajewska, H., 515
  - Galil, Z., 405~406, 487
  - Galler, B. A., 337
  - Games (博弈), 469
    - alpha-beta pruning ( $\alpha$ - $\beta$ 裁剪), 472~476
    - hash tables for (散列表), 207
    - minimax algorithm (极小极大算法), 469~472
  - Gap size in skip lists (跳跃表中的间隙容量), 535~536
  - Garbage collection (垃圾收集), 20
  - Garey, M.R., 406, 487
  - gcd (greatest common divisor) function (gcd(最大公约数)函数), 60~61, 64
  - General-purpose sorting algorithms (一般用途的排序算法), 300
  - Geometric series (几何级数), 4
  - getChainFromPrevMap function (getChainFromPrevMap函数), 365~366
  - getName function (getName函数), 190
  - Giancarlo, R., 487
  - Global optimums (全局最佳), 409
  - Godbole, S., 487
  - Goldberg, A.V., 406
  - Golin, M., 312
  - Gonnet, G.H., 182~183, 212, 259, 312
  - Graham, R.L., 41, 487
  - Grandchildren in trees (树的孙子), 114
  - Grandparents in trees (树的祖父), 114
  - Graph class (Graph类), 358~359

- Graphs (图), 339  
   bipartite (二分图), 398  
   breadth-first searches (广度优先搜索), 349~351  
   coloring (着色), 396  
   definitions for (定义), 339~340  
   depth-first searches (深度优先搜索), 见 Depth-first searches  
    $k$ -colorable ( $k$ 可着色的), 401  
   minimum spanning tree (最小生成树), 372~378  
   multigraphs (多重图), 401  
   network flow problems (网络流问题), 367~372  
   NP-completeness (NP完全性), 392~396  
   planar (平面图), 400  
   references for (引用), 404~407  
   representation of (表示), 340~342  
   shortest-path algorithms for (最短路径算法)  
     acyclic graph (无环图), 360~364, 483~384  
     all-pairs (所有顶点对), 364, 451~454  
     Dijkstra's algorithm (Dijkstra算法), 351~360  
   example (例子), 365~366  
   negative edge costs (负边值), 360~361  
   single source (单源点), 345~347  
   unweighted (未加权的), 347~352  
     topological sorts for (拓扑排序), 342~345  
     traveling salesman (旅行商), 480  
 Greatest common divisor (GCD) function (最大公约数 (GCD) 函数), 60~61, 64  
 Greedy algorithms (贪婪算法), 214, 409~410  
   approximate bin packing (近似装箱), 见 Approximate bin packing  
   for coin changing problem (硬币兑换问题), 410  
   Dijkstra's algorithm (Dijkstra算法), 352  
   Huffman codes (赫夫曼编码), 413~419  
   Kruskal's algorithm (Kruskal算法), 376~378  
   maximum-flow algorithms (最大流算法), 369  
   minimum spanning tree (最小生成树), 372~378  
   processor scheduling (处理器调度), 410~413  
 Gries, D., 41  
 Growth rate of functions (函数的增长率), 44~46  
 Gudes, E., 183  
 Guibas, L.J., 183, 212, 564  
 Gupta, R., 487  
**H**  
   .h files (.h文件), 15~16  
   Hagerup, T., 405  
   Haken, D., 486  
   Halting problems (停机问题), 392  
   Hamiltonian cycle (哈密尔顿回路), 388, 392~396  
   handleReorient function (handleReorient函数), 528, 532  
   Harary, F., 406  
   Harmonic numbers (调和数), 5  
   Harries, R., 212  
   hash function (hash函数), 186~188  
   hash\_map function (hash\_map函数), 204  
   hash\_set function (hash\_set函数), 204  
   Hash tables (散列表), 185  
     double hashing in (双散列), 199~200  
     extendible hashing (可扩散列), 204~205  
     hash function (散列函数), 186~188  
     linear probing in (线性探测), 193~195  
     overview (概论), 185~186  
     quadratic probing in (平方探测), 195~199  
     references for (引用), 211~212  
     rehashing for (再散列), 199~203, 514  
     separate chaining for (分离链接法), 188~192  
     in Standard Library (标准库), 204  
   Hasham, A., 259  
   HashEntry class (HashEntry类), 196~197  
   HashTable class (HashTable类), 189~190, 196~198  
   header files (头文件), 15~16  
   Header information, compilation of (头文件信息编译), 568  
   Header nodes for lists (表头结点), 84~85  
   Heap order, dequeues with (双端队列堆序), 514  
   Heap-order property (堆序性质), 216~217  
   Heaps (堆)  
     2-d (2-d堆), 561  
     binary (二叉堆), 见 Binary heaps  
     leftist (左式堆), 见 Leftist heaps  
     pairing (配对堆), 553~559  
     priority (优先堆), 见 Priority queues  
     skew (斜堆), 235~239, 497~499  
   Heapsort (堆排序)  
     analysis (分析), 272~274  
     comparing (比较), 306  
     implementation (实现), 270~272  
     references for (引用), 311  
   heapsort function (heapsort函数), 273~374  
   Heavy nodes in skew heaps (斜堆中的重结点), 498~499  
   Height of trees (树的高), 114  
     AVL (AVL树), 146  
     binary tree traversals (二叉树遍历), 159  
     complete binary (完全二叉), 215  
   Hibbard, T.H., 183, 312  
   Hibbard's increments (Hibbard增量), 269~270  
   Hiding information (信息隐藏), 12  
   Hirschberg, D.S., 487~488  
   Hoare, C.A. R., 312  
   Hoey, D., 488  
   Homomentric point set (同度点集), 479  
   Hopcroft, J.E., 69, 182, 336~337, 405~406



Horizontal arrays for skip lists (跳跃表的水平数组), 540  
 Horizontal links for AA-trees (AA树的水平链接), 542  
 Horvath, E.C., 312  
 Hu, T.C., 487  
 Huang, B., 312  
 Huffman, D.A., 487  
 Huffman codes (赫夫曼编码), 413~419, 485  
 Hulls, convex (凸包), 481~482  
 Hutchinson, J.P., 41  
 Hypotheses in induction (归纳假设), 6~7

## I

if/else statements in running time (if/else语句的运行时间), 51  
 Implementation/interface separation (实现/接口的分离), 15~17  
 Implicit type conversions (隐式类型转换)  
   with constructors (构造函数), 14  
   with pointers (指针), 296  
 Impossible problems (不可能解出的问题), 392  
 In-situ permutations (中间置换), 292  
 Incerpi, J., 312  
 #include preprocessor (#include预处理指令), 15~16  
 increaseKey function (increaseKey函数), 222  
 Increment sequences in Shellsorts (谢尔排序中的增量序列), 266~270  
 Indegrees of vertices (顶点的入度), 342  
 Indirect sorting (间接排序)  
   problems (问题), 295~297  
   process (处理), 292~294  
 Inductive proofs (归纳法证明)  
   process (处理), 6~7  
   recursion in (递归), 10~11  
 Infinite loop-checking programs (无限循环检测程序), 393  
 Infix to postfix conversion (中缀至后缀转换), 99~102  
 Information hiding (信息隐藏), 12  
 Information-theoretic lower bounds (信息理论的下界), 299  
 Inheritance for lists (表的继承), 85  
 init function (init函数), 90~91  
 Initializer lists for constructors (构造函数的初始化列表), 13~14  
 Inorder traversal (中序遍历), 121, 158  
 Input size in running time (运行时间中的输入大小), 47~49  
 insert function and insert operations (insert函数和插入操作)  
   2-d trees (2-d树), 549~552  
   AA-trees (AA树), 542~545  
   AVL trees (AVL树), 137~139  
     double rotation (双旋转), 141~149  
     single rotation (单旋转), 139~141  
   B-trees (B树), 162~164  
   binary heaps (二叉堆), 216~219  
   binary search trees (二叉查找树), 124~127, 129~130  
   binary searches (二分搜索), 59  
   binomial queues (二项队列), 242~244, 246, 492~497  
   d-heaps (d堆), 228  
   extendible hashing (可扩散列), 20  
   Fibonacci heaps (斐波那契堆), 505, 508  
   hash tables (散列表), 188~189, 192, 199  
   Huffman algorithm (赫夫曼算法), 419  
   iterators (迭代器), 77  
   leftist heaps (左式堆), 230~231, 234  
   linked lists (链表), 73~74  
   lists (表), 72, 85, 87, 92~93  
   maps (图), 166  
   multiway merges (多路合并), 302  
   pairing heaps (配对堆), 554, 557  
   priority queues (优先队列), 213~214  
   red-black trees (红黑树), 526~528, 532~533  
   sets (集), 165~166  
   skew heaps (斜堆), 237, 499  
   skip lists (跳跃表), 460~461, 535~536, 539  
   splay trees (伸展树), 153~155, 521, 523  
   treaps (treap树), 547~550  
 Insertion sorts (插入排序)  
   algorithm (算法), 262~263  
   analysis (分析), 264~265  
   comparing (比较), 306  
   implementation (实现), 263~264  
 insertionSort function (insertionSort函数), 263~265  
 insertionsortHelp function (insertionsortHelp函数), 264  
 Instantiation, explicit (显式实例化), 568~570  
 IntCell class (IntCell类)  
   constructors for (构造函数), 12~15  
   defaults for (默认值), 24~27  
   interface/implementation separation in (接口/实现的分离), 15~17  
   pointers in (指针), 19~21  
 Integers (整数)  
   greatest common divisors of (最大公约数), 60~61  
   multiplying (相乘), 438~439  
 Interface/implementation separation (接口/实现的分离), 15~17  
 Internal path lengths (内部路径长), 133  
 Inversion in arrays (逆序数组), 265~266  
 isActive function (isActive函数), 197~198  
 isEmpty function (isEmpty函数)  
   for binary heaps (二叉堆), 216  
   for binary search trees (二叉查找树), 126  
   for binomial queues (二项队列), 246



for leftist heaps (左式堆), 234  
 for top-down splay trees (自顶向下伸展树), 521  
 isLessThan function (isLessThan函数), 34~35  
 Isomorphic trees (同构树), 180  
 isPrime function (isPrime函数), 463  
 Iterators (迭代器), 75  
   const\_iterator, 77~79  
   for container operations (容器操作), 77  
   erase, 77  
   getting (获得), 76  
   for lists (表), 84~85, 87~88  
   for maps (图), 166~167  
   methods for (方法), 76~77  
   for sets (集), 165~166  
   stale (废物), 109  
 Iyengar, S.S., 564

**J**

Janson, S., 312  
 Johnson, D. B., 259, 406  
 Johnson, D. S., 406, 487  
 Johnson, S. M., 312  
 Jonassen, A. T., 183  
 Jones, D. W., 564  
 Josephus problem (Josephus问题), 108

**K**

$k$ -colorable graphs ( $k$ 可着色图), 401  
 $k$ -d trees ( $k$ -d树), 549~553, 563  
 Kaas, R., 259  
 Kaehler, E. B., 183  
 Kahn, A. B., 406  
 Karatsuba, A., 487  
 Karger, D. R., 406, 487  
 Karlin, A. R., 212  
 Karlton, P.L., 183  
 Karp, R. M., 212, 337, 405~406  
 Karzanov, A. V., 406  
 Kayal, N., 486  
 Kernighan, B. W., 41, 406  
 Kevin Bacon Game (Kevin Bacon游戏), 403  
 Keys (键)  
   in hashing (散列), 185~188  
   for maps (图), 166~170  
 Khoong, C. M., 259, 515  
 King, V., 406  
 Kitten Puzzle, 492  
 Klein, P.N., 406  
 Knapsack problem (背包问题), 396, 483  
 Knight's tour (马的环游), 483

Knuth, D. E., 41, 69, 183, 212, 259, 312, 337, 406, 487  
 Komlos, J., 212  
 Korsh, J., 486  
 Kruskal, J. B., Jr., 406  
 kruskal function (kruskal函数), 378  
 Kruskal's algorithm (kruskal函数), 376~378  
 Kuhn, H.W., 406

**L**

Labels for class members (类成员的标号), 12  
 Ladner, R. E., 259  
 Lajoie, J., 41  
 LaMarca, A., 259  
 Landau, G. M., 487  
 Landis, E. M., 182  
 Langston, M., 312  
 LaPoutre, J.A., 337  
 largeObjectSort function (largeObjectSort函数), 294~295  
 Larmore, L. L., 487  
 Last in, first out (LIFO) lists (后进先出表), 见Stacks  
 Lawler, E.L., 406  
 Lazy binomial queues (懒惰二项队列), 503~506  
 Lazy deletion (懒惰删除)  
   AVL trees (AVL树), 148  
   binary search trees (二叉查找树), 132  
   hash tables (散列表), 196  
   leftist heaps (左式堆), 255  
   lists (表), 109~110  
 Lazy merging (懒惰合并)  
   binomial queues (二项队列), 502~506  
   Fibonacci heaps (斐波那契堆), 500  
 Leaves in trees (树中的叶结点), 114  
 Lee, C.C., 488  
 Lee, D. T., 488, 564  
 Lee, K., 488  
 leftChild function (leftChild函数), 273  
 Leftist heaps (左式堆), 229  
   cutting nodes in (切除结点), 500~502  
   merging with (合并), 230~235  
   path lengths in (路径长), 229~230  
   references for (引用), 257  
   skew heaps (斜堆), 235~239  
 LeftistHeap class (LeftistHeap类), 234  
 LeftistNode structure (LeftistNode结构), 234  
 Lehmer, D., 456  
 Lelewer, D. A., 488  
 Lemke, P., 488  
 Lempel, A., 489  
 Length (长)  
   in binary search trees (二叉查找树), 133

- graph paths (图路径), 339
  - tree paths (树路径), 114
  - Lenstra, H. W., Jr., 488
  - Leong, H.W., 259,515
  - Level-order traversal (层序遍历), 159
  - Levels of AA-tree nodes (AA树结点层次), 541
  - Lewis, T.G., 212
  - L'Hôpital's rule (L'Hôpital法则), 45
  - Liang, F. M., 488
  - LIFO (last in, first out) lists (后进先出表), 见Stacks
  - Light nodes in skew heaps (斜堆中的轻结点), 498~499
  - Limits of function growth (函数增长率限制), 45
  - Lin, S., 406
  - Linear congruential generators (线性同余数发生器), 456~459
  - Linear-expected-time selection algorithm (线性期望时间选择算法), 290~291
  - Linear growth rate (线性增长率), 45~46
  - Linear probing (线性探测), 193~195
  - Linear worst-case time in selection problem (选择问题的线性最坏情形的运行时间), 435
  - Linked elements in skip lists (跳跃表中的链接元素), 535
  - Linked lists (链表), 73~74
    - circular (循环), 111
    - priority queues (优先队列), 214
    - skip lists (跳跃表), 459~461
    - stacks (栈), 96
  - Lippman, S. B., 41
  - List class (List类), 84~87
  - listAll function (listAll函数), 116
  - Lists (表), 72
    - adjacency (邻接), 341~342
    - arrays for (数组), 72
    - implementation of (实现), 83~94
    - linked (链表), 见Linked lists
    - queues (队列), 见Queues
    - skip (跳跃表), 459~461
    - stacks (栈), 见Stacks
    - in STL (STL), 74~75
    - vectors for (向量), 见Vectors
  - Load factor of hash tables (散列表的装填因子), 192~194
  - Local optimums (局部最优), 409
  - Log-squared growth rate (对数平方增长率), 45
  - Logarithmic growth rate (对数增长率), 45
  - Logarithmic running time (对数运行时间), 58
    - for binary searches (二分搜索), 58~60
    - for Euclid's algorithm (欧几里得算法), 60~61
    - for exponentiation (求幂), 61~62
  - Logarithms, formulas for (对数公式), 3
  - Logical representation for skip lists (跳跃表的逻辑表示), 537
  - Longest common subsequence problem (最长公共子序列问题), 482
  - Longest increasing subsequence problem (最长递增子序列问题), 482
  - Look-ahead factors in games (游戏中的向前因子), 471
  - Loops (环)
    - graph (图), 339
    - in running time (运行时间), 50
  - Lower bounds (下界)
    - of function growth (函数增长), 44
    - for sorting (排序), 265~266,297~299
  - Lueker, G., 212
- ## M
- M-ary search trees (M叉查找树), 160~161
  - Mahajan, S., 488
  - Main memory, sorting in (主存排序), 261
  - Majority problem (主元素问题), 68
  - makeEmpty function (makeEmpty函数)
    - for binary heaps (二叉堆), 216
    - for binary search trees (二叉查找树), 126, 132~133
    - for binomial queues (二项队列), 246
    - for hash tables (散列表), 189,191,198
    - for leftist heaps (左式堆), 234
    - for lists (表), 72
    - for top-down splay trees (自顶向下伸展树), 521,524
  - Manacher, G.K., 312
  - map class (map类), 113
  - Maps (图)
    - example (例子), 168~173
    - for hash tables (散列表), 185~186
    - implementation (实现), 167~168
    - operations (操作), 166~167
  - Margalit, O., 486
  - Martin, W.A., 564
  - Mathematics review (数学复习), 2
    - for algorithm analysis (算法分析), 43~46
    - exponents (指数), 3
    - logarithms (对数), 3
    - modular arithmetic (模运算), 5
    - proofs (证明), 6~7
    - recursion (递归), 7~11
    - series (级数), 4~5
  - Matrices (矩阵), 37
    - adjacency (邻接), 340~341
    - data members, constructors, and accessors for (数据成员、构造函数和访问函数), 37
    - destructors (析构函数), copy assignment (复制赋值), and copy constructors for (复制构造函数), 39
    - multiplying (相乘), 439~442,444~447
    - operator[] for, 37~39

- matrix class (matrix类), 38~39
  - Maurer, W.D. 212
  - max-heaps (max堆), 251
  - Maximum-flow algorithms (最大流算法), 367~372
  - Maximum subsequence sum problem (最大子序列和问题)
    - analyzing (分析), 47~49
    - running time of (运行时间), 52~58
  - MAXIT game (MAXIT游戏), 485
  - Maze generation (迷宫生成), 331~334
  - McCreight, E. M., 182,564
  - McDiarmid, C. J. H., 259
  - McElroy, M.D., 312
  - McKenzie, B.J., 212
  - Median-of-median-of-five partitioning (五分化中项的中项), 436~438
  - Median-of-three partitioning (三数中值分割法), 282, 285, 290
  - median3 function (median3函数), 285
  - Medians (中值)
    - samples of (例子), 435~437
    - in selectin problem (选择问题), 226
  - Melhorn, K., 183,212,405~406
  - Member functions (成员函数), 12
    - constant (常数), 15
    - signatures for (签名), 16
  - Memory (内存)
    - address-of operator for (地址运算符), 21
    - in computational models (计算模型), 46
    - sorting in (排序), 261
    - for vectors (向量), 80
  - Memory leaks (内存泄漏), 20,26~27
  - MemoryCell class (MemoryCell类)
    - compilation of (编译), 567~580
    - template for (模板), 30~31
  - merge function and merge operations (merge函数和合并操作)
    - binomial queues (二项队列), 246~249,492~297
    - d-heaps (d堆), 228
    - Fibonacci heaps (斐波那契堆), 500,508
    - leftist heaps (左式堆), 229~235
    - mergesorts (归并排序), 277
    - pairing heaps (配对堆), 554~556
    - skew heaps (斜堆), 497~499
  - mergeSort function (mergeSort函数), 276
    - analysis of (分析), 276~279
    - binomial queues (二项队列), 241~242,247~249
    - expression trees (表达式树), 123
    - external sorting (外部排序), 301~304
    - implementation of (实现), 274~276
    - multiway (多路), 302~303
    - polyphase (多相), 303~304
    - references for (引用), 311
    - skew heaps (斜堆), 235~239
  - Methods (方法), 12
  - Meyers, S., 41
  - Miller, G.L., 488
  - Miller, K. W., 488
  - Min-cost flow problems (最小值流问题), 372
  - min-heaps (最小堆), 251
  - Min-max heaps (最小最大堆), 254~255
  - Minimax algorithm (极小极大算法), 469~472
  - Minimum spanning trees (最小生成树), 372~373,480
    - Kruskal's algorithm (Kruskal算法), 376~378
    - Prim's algorithm (Prim算法), 373~376
    - references for (引用), 404
  - Modular arithmetic (模运算), 5,462~464
  - Moffat, A., 516
  - Molodowitch, M., 212
  - Monier, L., 488
  - Moore, J. S., 212
  - Moore, R. N., 487
  - Moret, B.M.E., 407,564
  - Morris, J. H., 212
  - Motwani, R., 488
  - Mulmuley, K., 488
  - Multigraphs (多重图), 401
  - Multiplying (相乘)
    - integers (整数), 438~439
    - matrices (矩阵), 439~442,444~447
  - Multiprocessors in scheduling problem (调度问题中的多处理器), 411~413
  - Multiway merges (多路合并), 302~303
  - Munro, J. I., 182,258~259,486,564
  - Musser, D. R., 41,312
  - Mutator functions (修改函数), 15
  - myHash function (myHash函数), 190
- ## N
- Negative-cost cycles (负值回路), 346~347
  - Negative edge costs (负边值), 360~361
  - Ness, D. N., 564
  - Nested loops in running time (运行时间中的嵌套循环), 50
  - Network flow (网络流), 367
    - maximum-flow algorithm (最大流算法), 367~372
    - references for (引用), 404
  - Networks, queues for (网络, 队列), 107
  - new operator (new运算符)
    - for pointers (指针), 19~20
    - for vectors (向量), 80
  - Newline characters (换行字符), 414~415
  - Next fit algorithm (下项适配算法), 421~422
  - Nievergelt, J., 183,212,565

Node class for lists (表的Node类), 84~85,87

Nodes (结点)

AA-trees (AA树), 541

binary search trees (二叉查找树), 124~125

binomial trees (二项树), 492

decision trees (决策树), 297

expression trees (表达式树), 121

leftist heaps (左式堆), 230~232,500~502

linked lists (链表), 73~74

lists (表), 84~85

pairing heaps (配对堆), 553

red-black trees (红黑树), 525

skew heaps (斜堆), 498~499

skip lists (跳跃表), 537

splay trees (伸展树), 510,518

treaps (treap树), 547

trees (树), 113~114

Nondeterministic algorithms (非确定型算法), 368

Nondeterminism (非确定性), 393,396

Nonpreemptive scheduling (非抢占调度), 410~413

Nonprintable characters (非打印字符), 414

Nonterminating recursive functions (非中止递归函数), 9

NP-completeness (NP完全性), 392

easy vs. hard (易与难), 392~393

NP class (NP类), 393~394

references for (引用), 404

traveling salesman problem (旅行商问题), 394~396

Null paths in leftist heaps (左式堆的零路径), 229~230,232

Null pointers (Null指针), 124

Null-terminator characters (空终止符), 27

numcols function (numcols函数), 38

numrows function (numrows函数), 38

## O

Object type (对象类型), 32~33

Objects (对象), 12

declaring (声明), 16~17

dynamic (动态), 19~20

function (函数), 34~35

passing (传递), 21~22

returning (返回), 22

Odlyzko, A., 183

Off-line algorithms (脱机算法)

approximate bin packing (近似装箱), 424~427

disjoint sets (不相交集), 316

Ofman, Y., 487

On-line algorithms (联机算法)

approximate bin packing (近似装箱), 420~421

definition (定义), 57

disjoint sets (不相交集), 316

One-dimensional circle packing problems (一维装圆问题), 480

One-parameter set operations (单参数集合运算), 165~166

oneCharOff function (oneCharOff函数), 169

Operands in expression trees (表达式树中的操作数), 121

Operators (操作符)

in expression trees (表达式树), 121~124

operator!=function (operator!=函数)

hash tables (散列表), 190~191

lists (表), 88~89

pointers (指针), 296

operator() function (operator()函数), 34~36

operator\* function (operator\*函数)

lists (表), 88~90

matrices (矩阵), 440~441

operator++ function (operator++函数), 88~90

operator< function (operator<函数)

in Comparable type (Comparable类型), 32

in Employee (Employee), 34

indirect sorting (间接排序), 294~295

pointers (指针), 294

sets (集), 166

operator<< function (operator<<函数), 32~33

operator= function (operator=函数), 24~26

binary search trees (二叉查找树), 126,133~134

binomial queues (二项队列), 246

in Employee (Employee), 32

iterators (迭代器), 76

leftist heaps (左式堆), 234

lists (表), 90~91

red-black trees (红黑树), 529

top-down splay trees (自顶向下伸展树), 521

vectors (向量), 80~81

operator== function (operator==函数)

hash tables (散列表), 190~191

lists (表), 88~89

operator[] function (operator[]函数)

lists (表), 75

maps (图), 166~167

matrices (矩阵), 37~39

vectors (向量), 75,80,82

overloading (重载), 32,34

Optimal binary search trees (最优二叉查找树), 447~451,485

optMatrix function (optMatrix函数), 448~449

Order (序)

binary heaps (二叉堆), 216~217

matrix multiplications (矩阵乘法), 444~447

Orlin, J.B., 405

O'Rourke, J., 488

Orthogonal range queries (正交范围查询), 552



- Othello game (奥赛罗游戏), 486
- Ottman, T., 182
- Overflow (溢出)
- in B-trees (B树), 164
  - in hash function (散列函数), 187
  - stack (栈), 102~103
- Overloading operators (重载操作符), 32,34,284~285
- Overmars, M.H., 337,565
- ## P
- pair class (pair类)
- for maps (图), 166
  - for sets (集), 165
- Pairing heaps (配对堆), 258,553~559,563
- Pan, V., 488
- Papadakis, T., 565
- Papadimitriou, C. H., 407
- Papernov, A. A., 312
- Paragraphs, right-justifying (右对齐段落), 481~482
- Parameters (参数)
- default (默认值), 12~13
  - passing (传递), 21~22
  - template (模板), 29
- Parentheses[] (圆括号)
- balancing (平衡), 96~97
  - for postfix conversions (后缀转换), 99~101
- Parents in trees (树中的父亲), 113~115
- Park, S. K., 488
- Parse trees (分析树), 174
- Partial match queries (部分匹配查询), 552
- partialSum function (partialSum函数), 49
- Partitions (划分)
- in 2-d trees (2-d树), 562
  - in quicksorts (快速排序), 280,282~285
  - in selection problem (选择问题), 436~438
- Passing parameters (传递参数), 21~22
- Patashnik, O., 41
- Paths (路径)
- augmenting (增长), 367~372
  - binary search trees (二叉查找树), 133
  - compression (压缩), 324~331
  - in directory structures (目录结构), 115~119
  - graph (图), 339
  - halving (平分), 336
  - leftist heaps (左式堆), 229~230,232
  - shortest (最短路径), 见Shortest path-algorithms
  - in trees (树), 114
- Pattern matching problem (字型匹配问题), 482~483
- percDown function (percDown函数), 273
- percolateDown function (percolateDown函数), 216, 221, 223~226
- Percolation strategies (过滤策略)
- for binary heaps (二叉堆), 217~225
  - for heapsorts (堆排序), 273
- Perlis, A. J., 183
- Peterson, W.W. 212
- Phases in Shellsorts (谢尔排序中的相), 266~267
- Pippenger, N., 212
- Pivots in quicksorts (快速排序中的枢纽元), 280~282
- place function (place函数), 468~469
- Planar graphs (平面图), 400,405
- Plane partitions in 2-d trees (2-d树的平面划分), 562
- Plauger, P. J., 41
- Plaxton, C. G., 312
- Plies (层), 471
- Poblete, P. V., 258
- pointer class (pointer类), 294
- Pointers (指针), 19~20
- in binary search trees (二叉查找树), 124~125
  - for defaults (默认), 26
  - for indirect sorting (间接排序), 292~297
  - for lists (表), 84
  - operations for (操作), 20~21
  - type conversions with (类型转换), 295~296
- Points, closest (最近点), 430~435
- Polygons, convex (凸多边形), 481
- Polyphase merges (多相合并), 303~304
- Poonen, B., 312
- pop function (pop函数)
- for priority queues (优先队列), 251
  - for stacks (栈), 94~96
- pop\_back function (pop\_back函数)
- for lists (表), 75,87
  - for stacks (栈), 96
  - for vectors (向量), 75,80,82~83
- pop\_front function (pop\_front函数)
- for lists (表), 75,85,87,90
  - for vectors (向量), 75
- Position of list elements (表元素的位置), 72
- Positive-cost cycles (正值回路), 362
- Postfix expressions (后缀表达式)
- infix conversion to (中缀到后缀表达式的转换), 99~102
  - stacks for (栈), 97~99
- Postorder traversal (后序遍历), 117~118,121,159
- Potential functions (位势函数), 495~497
- costs of (值), 513
  - for splay trees (伸展树), 511
- pow (power) functions (pow函数), 61~62
- Pratt, V. R., 212,313,486
- prefix codes (前缀码), 415~416
- Preorder traversal (前序遍历), 117,121,159
- Preparata, F.P., 488

Preprocessor commands (预处理器命令), 15~16

Prim, R.C., 407

Prim's algorithm (Prim算法), 373~376

Primary clustering in hash tables (散列表中的一次聚集), 193

Prime numbers (素数)

- probability of (概率), 62~63
- proofs for (证明), 7
- Sieve of Eratosthenes for (厄拉多塞筛), 67
- tests for (测试), 461~464

Prime table size for hash tables (散列表的素数表大小), 196

print function (print函数)

- in Employee (Employee), 32~33
- for iterators (迭代器), 78~79

Printable characters (可打印字符), 414

printCollection function (printCollection函数), 165

printHighChangeables function (printHighChangeables函数), 169

Printing (打印)

- queues for (队列), 106
- recursion for (递归), 10, 102~103

printList function (printList函数), 72~73

printOut function (printOut函数), 10

printPath function (printPath函数), 358

printRange function (printRange函数), 552

printTree function (printTree函数)

- for binary search trees (二叉查找树), 126
- for binary trees (二叉树), 158~159
- for red-black trees (红黑树), 528~529
- for top-down splay trees (自顶向下伸展树), 521

priority\_queue class (priority\_queue类), 251~252

Priority queues (优先队列), 213

- binary heaps for (二叉堆), 见Binary heaps
- $d$ -heaps ( $d$ 堆), 228~229
- for Dijkstra's algorithm (Dijkstra算法), 359
- for heapsorts (堆排序), 270~274
- in Huffman algorithm (赫夫曼算法), 419
- implementations of (实现), 214
- leftist heaps (左式堆), 229~235
- model for (模型), 213~214
- references for (引用), 258
- for selection problem (选择问题), 226~227
- for simulation (模拟), 227~228
- skew heaps (斜堆), 235~239
- in Standard Library (标准库), 251~252

Priority search trees (优先查找树), 563

private labels (private标号), 12

Probability distribution functions (概率分布函数), 107

Probing in hash tables (散列表中的探测)

- linear (线性探测), 193~195
- quadratic (平方探测), 195~199, 202~203

probRelPrime function (probRelPrime函数), 62~63

Processor scheduling problem (处理器调度问题), 410~413

Progress in recursion (递归中的推进), 9, 11

Proofs (证明), 6~7, 10~11

Proper ancestors in trees (树中的真祖先), 114

Proper descendants in trees (树中的真后裔), 114

Pruning (裁剪)

- alpha-beta ( $\alpha$ - $\beta$ ), 472~476, 486
- in backtracking algorithms (回溯算法), 464

Pseudorandom numbers (伪随机数), 455~459

public labels (public标号), 12

Puech, C., 564

Pugh, W., 488

push functions (push函数)

- for priority queues (优先队列), 251
- for stacks (栈), 94~96

push\_back function (push\_back函数)

- for lists (表), 75, 85, 87
- for stacks (栈), 96
- for vectors (向量), 75, 80, 82~83

push\_front function (push\_front函数)

- for lists (表), 75, 87
- for vectors (向量), 75

## Q

Quad trees (四叉树), 562

Quadrangle inequality (四边形不等式), 478

Quadratic growth rate (二次增长率), 45~46

Quadratic probing (平方探测)

- in hash tables (散列表), 195~199
- for rehashing (再散列), 202~203

Queries for 2-d trees (2-d树查询), 552~553

Queueing theory (排队论), 107

Queues (队列)

- applications (应用程序), 106~107
- array implementation (数组实现), 104~106
- binomial (二项队列), 见Binomial queues
- for breadth-first searches (广度优先搜索), 351
- for level-order traversal (层序遍历), 159
- model for (模型), 104
- priority (优先), 见Binary heaps, Priority queues
- simulating (模拟), 227~228
- for topological sorts (拓扑排序), 344~345

Quickselect algorithm (快速选择算法)

- implementation (实现), 290~291
- running time (运行时间), 435~437

Quicksort (快速排序)

- algorithm (算法), 279~280
- analysis (分析), 287~290

array size (数组大小), 284~285  
 comparing (比较), 306  
 partitions in (划分), 280,282~285  
 pivots in (枢纽元), 280~282  
 references for (引用), 312  
 routines for (例程), 284~286  
 for selection problem (选择问题), 290~291  
 quicksort function (quicksort函数), 285~286

## R

Rabin, M. O., 212,488  
 Raghavan, P., 488  
 Ramanan, P., 488  
 Random class (Random类), 457~458  
 Random collisions in hash tables (散列表中的随机冲突), 194  
 Random number generators (随机数生成器)  
   creating (创建), 455~459  
   references for (引用), 485~486  
 Random permutation generators (随机排列生成器), 65  
 Random pivot selections (随机枢纽元选择), 281~282  
 random0\_1 function (random0\_1函数), 457~458  
 randomInt function (randomInt函数), 457~458  
 Randomized algorithms (随机化算法), 454~455  
   primality tests (素性测试), 461~464  
   random number generators (随机数生成器), 455~459  
   skip lists (跳跃表), 459~461  
 Range queries (范围查询), 552~553  
 Ranks (秩)  
   binomial tree nodes (二项树节点), 492  
   for Fibonacci heaps (斐波那契堆), 508  
   in path compression (路径压缩), 325~331  
   for splay trees (伸展树), 510,512  
 Rao, S., 406  
 Rates of function growth (函数增长率), 44~46  
 read function (read函数)  
   in IntCell (IntCell), 12~13,17  
   in MemoryCell (MemoryCell), 31  
 reclaimMemory function (reclaimMemory函数)  
   for leftist heaps (左式堆), 234  
   for top-down splay trees (自顶向下伸展树), 521  
 Recurrence relations (递推关系)  
   in mergesorts (归并排序), 276~279  
   in quicksorts (快速排序), 287~290  
 Recursion (递归), 7~10  
   binary search trees (二叉查找树), 125  
   depth-first searches (深度优先搜索), 379  
   divide and conquer strategy (分治策略), 见Divide and conquer strategy  
   exponentiation (求幂), 61~62  
   induction (归纳法), 10~11  
   leftist heaps (左式堆), 232  
   maximum subsequence sum problem (最大子序列和问题), 54~58  
   mergesorts (归并排序), 275~279  
   path compression (路径压缩), 324~325  
   printing (打印), 10,102~103  
   quicksort (快速排序), 284  
   red-black trees (红黑树), 528  
   running time (运行时间), 51~52  
   selection problem (选择问题), 435~438  
   skew heaps (斜堆), 237~239  
   stack overflow from (栈溢出), 102~103  
   vs. tables (表), 442~444  
   tree definitions (树定义), 113  
   tree traversals (树遍历), 158~159  
 Recursively undecidable problems (递归不可判定问题), 393  
 Red-black trees (红黑树), 525  
   bottom-up insertion (自底向上插入), 526~527  
   references for (引用), 182  
   top-down deletion (自顶向下删除), 531,534  
   top-down insertion (自顶向下插入), 527~528  
 RedBlackTree class (RedBlackTree类), 530~533  
 Reed, B. A., 259  
 Reference (引用, 引址)  
   parameter passing by (通过引用参数传递), 21  
   returning values by (通过引用返回值), 22  
 Reference variables (引用变量), 23  
 Reflexive relations (自反关系), 315  
 Regions in 2-d trees (2-d树中的区域), 562  
 Registers for variables (变量寄存器), 102  
 rehash function (rehash函数), 203  
 Rehashing (再散列), 199~203,514  
 Reingold, E.M., 183,565  
 Relations in disjoint sets (不相交集的关系), 315  
 Relative growth rates of functions (函数的相对增长率), 44~46  
 Relative prime numbers, probability of (相对素数数目, 概率), 62~63  
 Relaxed heaps (松堆), 258  
 Remainder function (余数函数), 5  
 remove function (remove函数)  
   AA-trees (AA树), 546  
   binary heaps (二叉堆), 222  
   binary search trees (二叉查找树), 124,126~127,130~132  
   binomial queues (二项队列), 247  
   Fibonacci heaps (斐波那契堆), 500  
   hash tables (散列表), 189,191,197,199  
   SplayTree (伸展树), 524  
   top-down splay trees (自顶向下伸展树), 521  
   treaps (treap树), 550  
 Remove operations (除去操作), 见Delete operations



removeEveryOtherItem function (removeEvery-OtherItem函数), 78  
 Replacement selection in external sorting (外部排序中的替换选择), 304~305  
 reserve function (reserve函数), 75, 80~81, 83  
 Residual edges (残余边), 367  
 Residual graphs (残余图), 367~368  
 resize function (resize函数), 80~81  
 Return passing (返回传递), 22  
 Reverse Polish notation (逆波兰记法), 97  
 Right-justifying paragraphs (右对齐段落), 481~482  
 Rivest, R.L., 337, 486~487  
 Roberts, F. S., 41  
 Rohnert, H., 212  
 Roots (根)  
     decision trees (决策树), 297  
     leftist heaps (左式堆), 230~232  
     top-down splay trees (自顶向下伸展树), 518  
     trees (树), 113~114  
 rotate function (rotate函数), 528, 532  
 rotateWithLeftChild function (rotateWithLeftChild函数), 147~148, 521  
 rotateWithRightChild function (rotateWithRight-Child函数), 521  
 Rotation operations (旋转操作)  
     AVL trees (AVL树), 137~139  
         double (双), 141~149  
         single (单), 139~141  
     red-black trees (红黑树), 526~528  
     splay trees (伸展树)  
         limitations of (限制), 150~152  
         running times (运行时间), 509~512  
         top-down (自顶向下), 517~520  
         zig-zag (之字形), 152~158  
 Running times (运行时间), 49  
     in algorithm selection (算法选择), 1  
     amortized (摊还), 149  
     analysis checking for (分析检查), 62~63  
     definitions for (定义), 43~44  
     disjoint sets (不相交集), 321~322  
     disk I/O assumptions in (磁盘I/O假设), 159~160  
     divide and conquer algorithms (分治算法), 428~430  
     examples (例子), 49~50  
     factors in (因子), 46~49  
     general rules (一般法则), 50~52  
     logarithmic (对数的), 58~62  
     maximum subsequence sum problem (最大子序列和问题), 52~58  
     mergesorts (归并排序), 276~279  
     overestimating (过高评估), 63  
     quicksorts (快速排序), 279, 287~290

randomized algorithms (随机化算法), 454~455  
 rates of growth (增长率), 44~46  
 Shellsorts (谢尔排序), 268~270  
 skip lists (跳跃表), 459~461  
 splay trees (伸展树), 509~512  
 Runs in external sorting (外部排序的顺串), 301~305

## S

Sack, J.R., 258~259  
 Saini, A., 41  
 Saks, M. E., 337  
 Salesman problem (旅行商问题), 394~396, 480  
 Samet, H., 565  
 Samples of medians (中项样本), 435~437  
 Santoro, N., 258  
 Satisfiability problem (可满足性问题), 395~396  
 Saxe, J. B., 486  
 Saxena, N., 486  
 Schaffer, R., 313  
 Scheduling problem (调度问题), 410~413  
 Schonhage, A., 337  
 Schrage, L., 456, 488  
 Schwab, B., 564  
 Scoping operators (作用域操作符), 16  
 Scroggs, R.E., 183  
 Search trees (查找树)  
     AA-trees (AA树), 540~545  
     binary (二叉), 见Binary search trees  
     k-d trees (k-d树), 549~553, 563  
     red-black (红黑), 见Red-black trees  
     splay trees (伸展树), 见Splay trees  
     treaps (treap树), 547~550  
 Searches (搜索), 见Find operations  
     binary (二分), 58~60  
     breadth-first (广度优先), 349~351  
     depth-first (深度优先), 见Depth-first searches  
 Secondary clustering in hash tables (散列表中的二次聚集), 199  
 Sedgewick, R., 183, 258, 270, 312~313, 565  
 Seeds for random numbers (随机数的种子), 456  
 Seidel, R., 564  
 Selection problem (选择问题)  
     alternate algorithms for (交换算法), 1~2  
     divide and conquer strategy for (分治策略), 435~438  
     priority queues for (优先队列), 226~227  
     quicksorts for (快速排序), 290~291  
     references for (引用), 485  
 Selection replacement in external sorting (外部排序中的选择替换), 304~305  
 Self-adjusting structures (自调整结构)  
     binary search trees (二叉查找树), 136



- disjoint sets (不相交集), 见Disjoint sets
- lists (表), 110
- path compression (路径压缩), 334
- skew heaps (斜堆), 235~239
- splay trees (伸展树), 见Splay trees
- Sentinel nodes (标记结点), 84
- Separate chaining for hash tables(散列表中的分离链接法), 188~192
- Separate compilation of class templates (类模板中的分离编译), 35~37,567~568
  - explicit instantiation (显式实例化), 568~570
  - export directive (导出指令), 570
  - for header information (头文件信息), 568
- Separation of interface from implementation (接口与实现的分离), 15~17
- Sequences of random numbers (随机数序列), 455~456
- Series (级数), 4~5
- set class (set类), 113
- Sets (集)
  - disjoint (不相交集), 见Disjoint sets
  - implementation of (实现), 167~168
  - operations for (操作), 165~166
- setValue function (setValue函数), 32~33
- Shallow copies (浅复制), 25~26
- Shamos, M. I., 488
- Shapiro, H.D., 407,564
- Sharir, M., 407
- Shell, Donald L., 266~267,313
- Shellsort (谢尔排序)
  - comparing (比较), 306
  - description (描述), 266~268
  - references for (引用), 311
  - worst-case analysis of (最坏情形分析), 268~270
- shellsort function (shellsort函数), 267
- Shing, M. R., 487
- shortest function (shortest函数), 484
- Shortest-path algorithms (最短路径算法)
  - acyclic graphs (无环图), 360~364,483~484
  - all-pairs (所有点对), 364,451~454
  - Dijkstra's algorithm (Dijkstra算法), 351~360
  - example (例子), 365~366
  - negative edge costs (负边值), 360~361
  - single source (单源), 345~347
  - unweighted (无权的), 347~352
- Shrairman, R., 258
- Siblings in trees (树中的兄弟结点), 114~115
- Sieve of Eratosthenes (厄拉多塞筛), 67
- Signatures for member functions (成员函数的签名), 16
- Simon, I., 336
- Simple paths (简单路径), 339
- Simulation, priority queues for (优先队列模拟), 227~228
- Single rotation operations (单旋转操作)
  - in AVL trees (AVL树), 139~141
  - limitations of (限制), 150~152
- Single source algorithm for shortest-path problem (最短路径的单源点算法), 345~347
- Sinks in network flow (网络流中的汇点), 367
- size function and size (size函数和大小)
  - arrays (数组), 27,105~106
  - binomial queues (二项队列), 240
  - containers (容器), 75
  - directory structures (目录结构), 118~119
  - hash tables (散列表), 185~186,196
  - input, in running time (输入, 运行时间), 47~49
  - lists (表), 85~86
  - maps (图), 166
  - sets (集), 165
  - vectors (向量), 18~19,80,82~83
- skew function (skew函数), 542~546
- Skew heaps (斜堆), 235~239
  - amortized analysis of (摊还分析), 497~499
  - references for (引用), 257
- Skiena, S.S., 488
- Skip lists (跳跃表)
  - deterministic (确定性), 535~539
  - working with (运行), 459~461
- SkipNode class (SkipNode类), 537~538
- Slack time in acyclic graphs (无环图中的松弛时间), 363~364
- Sleator, D. D., 183,258~259,516,565
- Smart pointer classes (智能指针类), 295
- Smart union algorithms (灵巧求并算法), 321~323
- Smith, H. F., 183
- Smith, W. D., 488
- Smolka, S. A., 487
- sort function (sort函数), 261~262
- Sorting (排序)
  - algorithm comparison (算法比较), 305~306
  - bucket sorts (桶排序), 299~300
  - external (外部), 300~305
  - heapsorts (堆排序), 270~274
  - indirect (无向的), 292~207
  - insertion sorts (插入排序), 262~265
  - lower bounds for (下界), 265~266,297~299
  - mergesorts (归并排序), 274~279
  - quicksort (快速排序), 见Quicksort
  - references for (引用), 311~313
  - Shellsort (谢尔排序), 266~270
  - topological (拓扑的), 342~345
- Sources in network flow (网络流中的源), 367
- Spanning trees, minimum (最小生成树), 372~372,480
  - Kruskal's algorithm (Kruskal算法), 376~378

- Prim's algorithm (Prim算法), 373~376  
 references for (引用), 404
- Sparse graphs (稀疏图)  
 adjacency lists for (邻接表), 341  
 with Dijkstra's algorithm (Dijkstra算法), 358
- Special cases in recursion (递归中的特殊情形), 9
- Spelling checkers (拼写检查程序), 207
- Spencer, T.H., 406
- splay function (splay函数), 522
- Splay trees (伸展树), 136, 149~150  
 amortized analysis (摊还分析), 509~513  
 vs. single rotations (与单旋转), 150~152  
 top-down (自顶向下), 517~524  
 zig-zag rotations in (之字形旋转), 152~158
- SplayTree class (SplayTree类), 521~524
- split function (split函数), 542~546
- stable\_sort function (stable\_sort函数), 262
- Stable sorting algorithms (稳定的排序算法), 262, 309
- Stack frames (栈帧), 102
- Stacks (栈)  
 for balancing symbols (平衡符号), 96~97  
 for function calls (函数调用), 102~104  
 implementation (实现), 95~96  
 for infix to postfix conversions (中缀至后缀的转换), 99~102  
 model of (模型), 94~95  
 for postfix expressions (后缀表达式), 97~99  
 for topological sorts (拓扑排序), 344
- Stale iterators (废物迭代器), 109
- Standard Template Library (STL) (标准模板库)  
 containers in (容器), 74~75  
 hash tables in (散列表), 204  
 insertion sort implementation (插入排序实现), 263~264  
 maps in (图), 166~167  
 priority queues in (优先队列), 251~252  
 sets in (集), 165~166  
 sorting algorithms in (排序算法), 261~262
- Stasevich, G. V., 312
- Stasko, J. T., 565
- States in decision trees (决策树中的状态), 297
- Steiglitz, K., 407
- Stephenson, C. J., 565
- Stirling's formula (Stirling公式), 309
- STL, 见Standard Template Library (STL)
- Strassen, V., 488
- Strassen's algorithm (Strassen算法), 439~442
- strcpy function (strcpy函数), 27
- string class (string类), 17~19, 27
- Strings, C-style (C风格字符串), 26~29
- Strip areas in closest points problem (最近点问题的带状区域), 431~432
- strlen function (strlen函数), 27
- Strong, H.R., 212
- Strong components (强分支), 390~391
- Strongly connected graphs (强连通图), 339~340
- Strothotte, T., 258~259
- Stroustrup, B., 41
- struct keyword for lists (表的struct关键字), 85
- Suboptimal solutions (次最优解), 409
- Substitution problem, maps for (置换问题的图), 168~173
- Subtraction of pointers (指针减少), 297
- Successor positions in games (游戏中的后继位置), 469
- Suel, T., 312
- Sums (和, 求和)  
 maximum subsequence sum problem (最大子序列和问题), 47~49, 52~58  
 telescoping (叠缩), 278, 289
- swapchildren function (swapchildren函数), 234
- Symbol tables (符号表), 207
- Symbols in header files (头文件中的符号), 15~16
- Symmetric relations in disjoint sets (不相交集对称关系), 315
- System clocks for random numbers (随机数的系统时钟), 456
- Szemerédi, E., 212
- ## T
- Table size in hash tables (散列表中的表大小), 185~186, 196
- Tables (表)  
 vs. recursion (递归), 442~444  
 symbol (符号), 207  
 transposition (置换表), 207, 471
- Tail nodes (尾结点), 84~85
- Tail recursion (尾递归), 102~103, 125
- Takaoka, T., 488
- Tapes, sorting on (排序上的磁带), 261, 300~305
- Tardos, E., 406
- Tarjan, R. E., 183, 212, 258~259, 337, 405~407, 486, 515~516, 565
- Telescoping sums (叠缩求和), 278, 289
- Templates (模板), 29  
 class (类), 30~32  
 Comparable and Object (Comparable和Object), 32~33  
 compilation of (编译), 35~37, 567~568  
 explicit instantiation (显式实例化), 568~570  
 export directive (导出指令), 570  
 for header information (头文件信息), 568  
 function (函数), 29~30
- Terminal positions in games (游戏中的终端位置), 469, 471
- Tests, primality (素性测试), 461~464
- Theta notation ( $\Theta$ 记法), 43~46

- Thornton, C., 183  
 Threaded trees (线索树), 168,180  
 Threads (线索), 180  
 Three-parameter insertion sorts (三参数插入排序), 263,265  
 Thurston, W. P., 183  
 Tic-tac-toe game (三连游戏), 469~472  
 Ticks for event simulation (事件模拟的滴答), 227  
 Time bound proof for Fibonacci heaps (斐波那契堆的时间界证明), 506~509  
 Top-down red-black trees (自顶向下红黑树), 527~528,534  
 Top-down splay trees (自顶向下伸展树), 517~524, 563  
 Top of stacks (栈顶), 94~95  
 top operations (栈顶操作)  
     priority queues (优先队列), 251  
     stacks (栈), 94~95  
 Topological sorts (拓扑排序), 342~345  
 topSort function (topSort函数), 344~345  
 Traffic flow, graphs for (交通图), 340  
 Traffic problems (交通问题), 410  
 Transitive relations (传递关系), 315  
 Transposition tables (置换表), 207,471  
 Traveling salesman problem (旅行商问题), 394~396,480  
 Traversing (遍历)  
     binary trees (二叉树), 158~159  
     directories (目录), 115~119  
 Treap class (Treap类), 548~550  
 TreapNode structure (TreapNode结构), 548  
 Treaps (treap树), 547~550  
 Trees (树)  
     2-3 (2-3树), 515  
     2-d (2-d树), 562  
     AA (AA树), 540~545  
     AVL (AVL树), 136~139  
         double rotation (双旋转), 141~149  
         single rotation (单旋转), 139~141  
     B-trees (B树), 159~164  
     binary (二叉树), 119~124  
     Cartesian (笛卡儿树), 563  
     decision (决策树), 297~299  
     definitions (定义), 113~114  
     for disjoint sets (不相交集), 318  
     game (博弈), 472~473  
     implementations of (实现), 114~115  
     isomorphic (同构的), 180  
     k-d ( $k$ -d), 549~553  
     minimum spanning (最小生成树), 372~373,480  
         Kruskal's algorithm (Kruskal算法), 376~378  
         Prim's algorithm (Prim算法), 373~376  
     parse (分析树), 174  
     quad (四叉树), 562  
     red-black (红黑树), 525  
         bottom-up insertion (自底向上插入), 526~527  
         top-down deletion (自顶向下删除), 531,534  
         top-down insertion (自顶向下插入), 527~528  
     splay (伸展), 136,149~150  
         amortized analysis (摊还分析), 509~513  
         vs. single rotations (单旋转), 150~152  
     top-down (自顶向下), 517~524  
     zig-zag rotations in (之字形旋转), 152~158  
     threaded (线索), 168,180  
     traversing (遍历), 115~119,158~159  
     treaps (treap树), 546~550  
     weight-balanced (加权平衡的), 563  
 Tries (检索树), 414~419  
 Tsur, S., 183  
 Tucker, A., 41  
 Turing machines (图灵机), 396  
 turnpike function (turnpike函数), 466~467  
 Turnpike reconstruction problems (收费公路重建问题), 465~469  
 2-3 trees (2-3树), 515  
 2-d heaps (2-d堆), 561  
 2-d trees (2-d树), 562  
 Two-dimensional range queries (二维范围查询), 550~551  
 Two-parameter operations (两参数操作)  
     insertion sorts (插入排序), 263~264  
     sets (集), 165~166  
 Two-pass algorithms (两趟算法)  
     Huffman algorithm (赫夫曼算法), 419  
     pairing heap merging (配对堆合并), 555,558~559  
 Type conversions (类型转换)  
     implicit (隐式), 14  
     with pointers (指针), 295~296
- ## U
- Ullman, J. D., 69,182,336~337,405,487  
 Unary minus operator (一元减运算符), 121  
 Undecidable problems (不可判定问题), 392  
 Undirected graphs (无向图), 339  
     biconnected (双连通), 381~385  
     depth-first searches (深度优先搜索), 379~381  
 Uninitialized pointers (未初始化指针), 19  
 Union algorithms (求并运算), 321~323  
 Union-by-height approach (按高度求并方法), 322~323,325  
 Union-by-rank approach (按秩求并方法), 325~331  
 Union-by-size approach (按大小求并方法), 321~323,325  
 Union/find algorithm (求并/查找算法)  
     analysis (分析), 326~331  
     for disjoint sets (不相交集), 316  
 union operations (求并运算)  
     disjoint sets (不相交集), 316~321  
     Kruskal's algorithm (Kruskal算法), 377

unionSets function (unionSets函数), 320,323~324  
 unweighted function (unweighted函数), 350~352  
 Unweighted path length (未加权路径长), 345,347~352  
 Upper bounds of function growth (函数增长率上界), 44

## V

Vallner, S., 259  
 Values (值)  
   parameter passing by (按值参数传递), 21  
   returning (返回), 22  
 van Emde Boas, P., 259  
 van Kreveld, M. J., 337  
 van Leeuwen, J., 337,565  
 van Vleith, A., 488  
 Variables (变量)  
   reference (引用), 23  
   stacks for (栈), 102  
 vector class (vector类), 17~19,27,80~83  
 Vectors (向量)  
   for adjacency lists (邻接表), 341  
   implementation (实现), 79~83  
   iterators for (迭代器), 75~79  
   for stacks (栈), 95  
   in STL (STL), 74~75  
 Vertex class (Vertex类), 341,358  
 Vertex cover problem (顶点覆盖问题), 404  
 Vertices (顶点)  
   graph (图), 339~340  
   topological sorts for (拓扑排序), 342~345  
 Vishkin, U., 487  
 Visibility in classes (类中的可见性), 12  
 Vitter, J. S., 212, 565  
 Voronoi diagrams (Voronoi图), 481  
 Vuillemin, J., 259,516,565

## W

Wagner, R. A., 489  
 Weakly connected graphs (弱连通图), 340  
 Wegman, M. N., 212  
 weight-balanced trees (加权平衡树), 182, 563  
 Weighted path lengths (加权路径长), 345~346,351~360  
 weightedNegative function (weightedNegative函数), 361  
 Weights (权)  
   graph edges (图边), 339~341

  in Huffman algorithm (赫夫曼算法), 416~417  
 Wein, J., 486  
 Weiss, M.A., 41,258,313,564  
 Westbrook, J., 337  
 Williamms, J. W. J., 259,313  
 Winograd, S., 487  
 witness function (witness函数), 463~464  
 Witten, I.H., 486  
 Wong, C.K., 564  
 Wong, J.K., 406  
 Wood, D., 182  
 Word puzzles (字谜), 1  
   hash tables for (散列表), 208  
   word ladders (字梯), 365~366  
   word substitution problem (字替换问题), 168~173  
 Worst-case analysis (最坏情形分析), 47  
   quicksorts (快速排序), 287  
   randomized algorithms (随机化算法), 454~455  
   Shellsort (谢尔排序), 268~270  
   union-by-rank approach (按秩求并方法), 325~331  
 write function (write函数)  
   in IntCell (IntCell), 12~13,17  
   in MemoryCell (MemoryCell), 31

## Y

Yao, A. C., 183,212,313,337,407,489  
 Yao, F. F., 489

## Z

Zero-slack edges (零松弛边), 364  
 Zhang, Z., 489  
 Zig operations (单旋转操作)  
   for red-black trees (红黑树), 526~627  
   for splay trees (伸展树), 509~512,517~518,520  
 Zig-zag operations (之字形操作)  
   for red-black trees (红黑树), 526~527  
   for splay trees (伸展树), 152~158,509~512,517~520  
 Zig-zig operations (一字形操作)  
   for red-black trees (红黑树), 526  
   for splay trees (伸展树), 509~512,517~520  
 Zijlstra, E., 259  
 Ziv, J., 489  
 Ziv-Lempel encoding (Ziv-Lempel编码), 485  
 Ziviana, N., 182